UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

Degree in Computer Engineering

TRABAJO FIN DE GRADO

BACHELOR'S THESIS

DESARROLLO DE UN MOTOR MODULAR PARA DESARROLLO DE VIDEOJUEGOS 2D

DEVELOPMENT OF A MODULAR ENGINE FOR THE CREATION OF 2D VIDEOGAMES

Arcadio Garcia Salvadores

Tutor: Carlos Aguirre Maeso

May 2017

Desarrollo de un motor modular para desarrollo de videojuegos 2D

DEVELOPMENT OF A MODULAR ENGINE FOR THE CREATION OF 2D VIDEOGAMES

AUTOR: Arcadio Garcia Salvadores

TUTOR: Carlos Aguirre Maeso

Escuela Politécnica Superior

Universidad Autónoma de Madrid

May 2017

Resumen

El desarrollo de videojuegos es un campo de la ingeniería de software que ha crecido en complejidad e importancia económica durante las últimas décadas, y los motores de videojuegos se han convertido en una solución popular para reusar código entre proyectos y proveer una capa de abstracción.

Este Trabajo de Fin de Grado describe el diseño y la implementación de Clockwork, una platforma modular para el desarrollo de videojuegos, que consiste de un micromotor, un gestor de paquetes, un runtime, una API estándar para bibliotecas de renderizado, unas herramientas de línea de comandos, una extensión de Visual Studio Code y un conjunto de puentes para exportar los juegos.

La Plataforma Clockwork es completamente open-source y modular, permitiendo reemplazar y personalizar cualquier modulo para adaptarse a cualquier necesidad especifica de un juego, mientras que promueve que el desarrollador escriba código reusable y elegante usando programación orientada a eventos, herencia y composición. Hace posible a los desarrolladores compartir fácilmente su código con otros gracias al gestor de paquetes, y proporciona una experiencia sencilla para el desarrollador con sus propias herramientas. Todo esto esta implementado con tecnología web estándar, usando JavaScript y formatos estándar como JSON, permitiendo que puedan expandir Clockwork escribiendo herramientas adicionales y portándolo a otras plataformas si así lo desean.

La Plataforma Clockwork ha sido publicada como open-source software en GitHub, y ha sido usada exitosamente para desarrollar diferentes juegos, demos, bibliotecas de renderizado y paquetes que demuestran la versatilidad de su diseño. La página web http://clockwork.js.org contiene más información sobre el proyecto.

Abstract

Game development is a field of software engineering that has grown in complexity and economical importance over the last decades, and game engines have become mainstream as a solution for reusing code across projects and providing a layer of abstraction.

This Bachelor Thesis describes the design and implementation of Clockwork, a modular platform for the development of videogames, that consists of a microengine, a package manager, a runtime, a standard API for rendering libraries, CLI tools, a Visual Studio Code extension and a set of bridges for exporting the games.

The Clockwork Platform is completely open-source and modular, thus allowing to replace and customize any module to adapt to game-specific needs, while it encourages the developer to write reusable and elegant code using event-oriented programming, inheritance and composition. It allows developers to easily share their code with others thanks to the package manager, and provides an end-to-end developer friendly experience with its own tools. All of this is implemented on standard web tech, using vanilla JavaScript and standard formats such as JSON,

empowering developers to further expand Clockwork by writing additional tooling and porting it to new platforms as they please.

The Clockwork Platform has been published as open-source software on GitHub, and has been used successfully to develop different games, demos, rendering libraries and packages that showcase the versatility of its design. More information about the platform can be found at http://clockwork.js.org.

Palabras clave

JavaScript, HTML5, Motor de videojuegos, Desarrollo de videojuegos, Videojuegos, Web, Universal Windows Platform, Node.js, NPM, Visual Studio Code

Keywords

JavaScript, HTML5, Game engine, Game development, Videogames, Web, Universal Windows Platform, Node.js, NPM, Visual Studio Code

*Any application that **can** be written in JavaScript, **will** eventually be written in JavaScript.*

Jeff Atwood

Acknowledgments

I would like to thank my colleague Silvia Barbero for being the first adopter of the technology developed on this work and inspiring me to push the platform further.

I also want to thank my tutor Carlos Aguirre for his support and help reviewing this work.

Finally, I would like to thank the Microsoft Student Partner program, for providing me with the means to host the cloud-based infrastructure of this project, and the Microsoft Edge team for sending me the Raspberry Pi used to test the engine on IoT devices.

Table of Contents

Table of Figures

INTRODUCTION

This section will explain the motives for this Bachelor Thesis, the objectives proposed and the structure of this document.

Motivation

The video game industry is a growing economic sector that back in 2012 already reached $79 billion in revenue [1]. It employs thousands of people across dozens of different job disciplines [2], and a considerable percentage of them are programmers.

However, most videogames are not developed from scratch, relying instead on some sort of middleware. The growing complexity of the games being developed requires more powerful tools, and game engines aim to provide a set of higher level tools that can be easily reused across titles and, in some cases, abstract the game from the underlying platform (Operating System, Graphics API…) to make it platform independent.

Some game engines are in-house tools developed by game studios, with a specific game genre and requirements in mind, while others are popular tools used widely across the industry. For example, games made with Unity had 5 billion downloads in Q3 2016 and are running in 2.4 billion mobile devices [3]. The huge market share of some of this products doesn't mean that it is not a competitive market, since all engines have its pros and cons: some target indie games (developed by amateurs and independent teams) and others are designed for AAA games (blockbuster games with the highest budgets), some provide a layer of abstraction while others strive to be as efficient as possible, and of course they are developed on top of different technologies and provide APIs and/or scripting capabilities in different programming languages.

The motivation and goal of this work is to develop a game engine that tries to differentiate itself by breaking with some of the traditional conventions in game engine design by instead following some popular trends in software development:

- The rise of the Web as a platform for developing complex applications.
- The growing popularity of package managers that are bundled with programming runtimes (like NPM for Node.js, NuGet for .NET, RubyGems for Ruby…).
- The availability of cross-platform development solutions (like Xamarin, Apache Cordova…) that allow developers to target many platforms with the same code, while still allowing to provide OS-specific code via 'plugins'.

Objectives

The objective of this work is to design and develop a game engine that satisfies the following requirements:

- It should allow game developers to code, debug, run and publish games using JavaScript as the programming language.
- The games developed should use standard web technologies (HTML5 and JavaScript) and therefore run on any modern web browser, but they should also be able to be easily packaged as native apps for a specific platform.
- The engine should provide a simple API for common functionality, such as rendering, collision detection or processing input, but also allow developers to extend and modify those capabilities according to their needs.
- The engine should provide a way to encapsulate and reuse code easily.
- Developers should be able to easily manage their dependencies, and publish their own packages.
- Tooling should be made available that allows developers to seamlessly run, debug and publish their apps.

Structure of this document

In the *State of the art* section, the features and limitations of currently used game engines are discussed. Then, the *Design* section describes the functionality of the software developed, and the *Development* section details the techniques and technologies used to implement it. Finally, the *Integrations, tests and results* section lists the games and demos developed on top of the platform to showcase its potential, and the *Conclusions and future work* section sums up the project and defines what the next steps to further improve the platform would be.

STATE OF THE ART

As mentioned before, there is a large number of game engines, but for the purpose of this work these two are the most interesting:

- **Unity**: Unity is one of the most popular game engines nowadays, being used by 34% of the top 1000 free mobile games [3]. It is platform independent and provides its own easy to use editor, making it is especially popular with small teams and for use in the prototyping stage.
- **Phaser**: Phaser is a HTML5 game engine for 2D games that provides a comprehensive set of features, making it the most popular library for developing games for the web.

Other popular engines and platforms are not considered because they don't support the web as a target platform (like Unreal Engine) or are discontinued (like Adobe Flash).

There is a clear divide between "native" game engines, like Unity, and "web" engines, like Phaser. Native engines provide a lot of tools common in native software development, such as debuggers, package managers, editors… and the ability to easily export games as native apps, while web engines are usually open source and built on standard web tech, at the cost of offering little to no tooling.

There is another common characteristic, even in modern open-source engines: they usually follow a monolithic approach, where the engine and all of its features are bundled together. They might offer a few options (like Phaser, which lets the developer choose between different physics models, or Unity, that allows to customize the rendering pipeline to some extent), but they restrict the developer from modifying or even replacing every aspect of core aspects of the engine such as the rendering library, the input processing or the physics.

The game engine developed here will try to combine the benefits from both "web" and "native" engines, while at the same time following a modular approach that allows developers to completely customize every subsystem.

DESIGN

In order to provide all the desired functionality, instead of following the traditional monolithical approach (where the engine is a huge library that bundles all the functionality) the engine developed here will follow a modular approach, and will come with a set of tools that will be an integral part of the development experience.

Since this approach resembles a general-purpose software platform more than traditional game engines, from now on the term platform will be used to describe the whole solution, and the term engine will be reserved just for the library around which all the games will be built. The name Clockwork has been chosen for the platform as a metaphor for many different pieces working in unison.

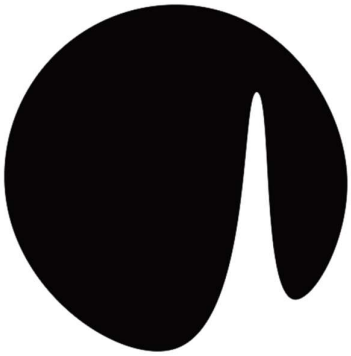


Figure 1: Logo of the Clockwork Platform

The Clockwork platform is composed of the following parts:

- **ClockworkCore**: This is the game engine, it provides just the basic functionality that every game will need.
- **Components**: These are the basic code units, they describe how a game object behaves and can be written by the developer or downloaded from the Clockwork Package Manager.
- **Clockwork Package Manager**: Like other package managers such as NPM or NuGet, it allows developers to download components, declare dependencies and share their code.
- **ClockworkRuntime**: It is the environment in which the code is executed, it provides a layer of OS abstraction and provides the necessary APIs for the development tools to attach to.

- **Rendering libraries**: They are libraries that take care of rendering the game. The default one is Spritesheet.js, a general purpose 2D game library, but it is very easy to develop shims to use any other library (Three.js, Babylon.js, Pixi.js…).

- **Clockwork CLI tools**: A command line tool, distributed through NPM, that allows developers to work on their games. It can be paired with any editor/IDE.

- **Visual Studio Code Clockwork Extension:** This is the most feature-rich tool for building Clockwork games, it builds on top of the CLI tools and provides additional functionality such as debugging.

- **Bridges**: These are tools that export games as native apps

Here it is a detailed overview of each of them:

ClockworkCore

ClockworkCore is a hybrid between a minimal game engine and a language runtime: on one hand it provides functionality like management of game levels and collision detection and on the other it also provides programming language-like features like inheritance, composition and message passing.

Some of the functionality that is provided as part of the most popular engines (physics, rendering, audio, networking) is not part of Clockwork Core, since it aims to be as versatile and flexible as possible, to let developers choose between many alternatives for those features or build their own if they need to, and they can find the official alternatives in the Clockwork Package Manager repository instead.

Components

Components are pieces of code that define how objects in the game should behave, they specify a spritesheet (which is a data structure that holds all the relevant information for rendering), define event handlers and collision shapes and can be combined through composition and inheritance.

Clockwork Package Manager

The Clockwork Package Manager is the combination of a tool and an online repository that allows developers to upload and download packages (that may contain components, collision

detectors…) and manage dependencies in Clockwork projects. For example, developers can use the gamepad component in their games to be able to process Xbox controller input, just by typing clockwork add gamepad in their terminal.

The decision to ship many of the features other game engines provide in the engine itself as published packages instead is due to the fact that it allows to easily keep the engine updated and provide new functionality, and developers will be able to replace them with their own packages. In the long run, the hope is that this will empower developers to create packages that can be reused by the community and will help to create a vibrant ecosystem similar to other platforms like Node.js/NPM.

Clockwork Runtime

Clockwork Runtime is a native program that hosts ClockworkCore and runs the games, providing the APIs for loading the game packages, debugging them and providing additional functionality like the live level editor. When the game is packaged for distribution, a lightweight version of the runtime will be attached to it, allowing the developers to debug the games even after they are packaged for release, and enabling support for modding with very little effort made by the game developer.

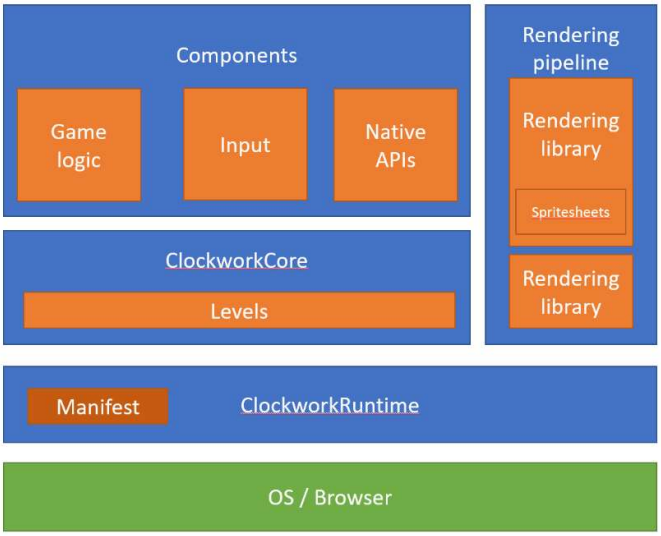


Figure 2: Diagram showing how Clockwork Runtime integrates other components

Rendering libraries

ClockworkCore doesn't take care of the rendering, it instead provides a way to register rendering libraries that will perform that function, which just need to expose a simple standardized API.

This approach is very flexible, and custom libraries have been developed to support the following scenarios:

- Rendering through the 2D Canvas API (for 2D games).
- Rendering through the 3D WebGL API (for 3D games).
- Rendering as HTML DOM elements.
- Rendering as text strings (for a screenless device that will read them aloud, or for retro text-only games).
- Recording the rendering data for 'replaying' it later.

And it is also possible to use simple 'proxy' libraries to:

- Send the data over a connection to another device that will render the results (useful for augmented reality scenarios, mirroring the game to other devices, sending data to a Bluetooth device…)
- Send the data to any of the previous libraries, allowing:
    - Adaptative experiences, that display the exact same game differently depending on the device capabilities (e.g. a game that renders on 2D or 3D depending on the graphical power of the device, or switches to voice mode if there is no screen) or the user limitations (e.g. a game that switches from a 'full' graphical experience to custom rendering libraries adapted for people with daltonism, epilepsy or other visual impairments, or even to a 'read aloud' mode for blind people).
    - Multiplayer experiences, rendering the game in many devices at once.
    - Rendering and recording the game as the same time.

If the developer doesn't specify otherwise, ClockworkRuntime will use the default 2D canvas animation library (Spritesheet.js), which aims to provide all the functionality a 2D spritesheet-based game will need.

CLI tools

clockwork-tools is a multiplatform command line tool that allows developers to build games with their favorite text editor/IDE. It is available via NPM and has the following features:

- Creating a starter Clockwork project (the corresponding command is *clockwork init*).
- Package the game into a .cw file, that can be loaded into the Clockwork Runtime (*clockwork build*).
- List the modules in the CWPM repository and the available versions of each one *(clockwork list*), add them as a dependency to the project (*clockwork add*), and update the dependencies (*clockwork update*).
- Create an account in the CWPM repository, that will be used to upload packages (*clockwork register*).
- Publish modules to the CWPM repository and update them (*clockwork publish*).

The npm package also exposes a JavaScript API for performing those tasks, so developers can incorporate these features into their favorite tools. There is already a Visual Studio Code extension that uses this package, which is the recommended way to build Clockwork games.

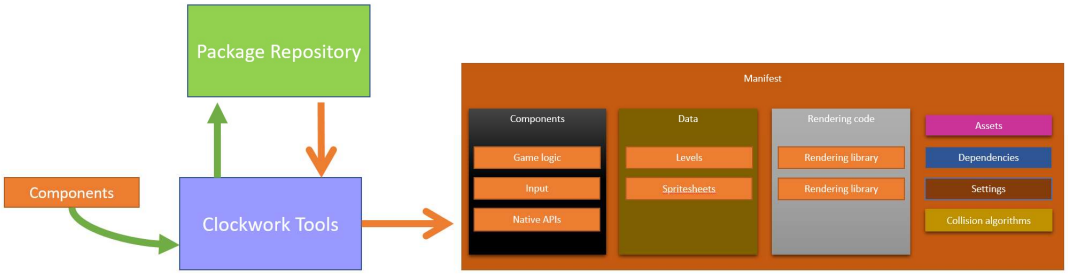


Figure 3: Diagram showing how the CLI tool can upload and add dependencies to packages

Visual Studio Code extension

There is an extension named 'clockwork' in the Visual Studio Code extension marketplace, that allows developers in any platform to build games using professional grade tools. It provides the following functionality:

- Creating an starter Clockwork project (F1 > *Create Clockwork Project*).

- Packaging the project into a .cw file (F1 > *Build Clockwork Project*).
- Packaging a project and deploying it to the runtime (F1 > *Deploy Clockwork Project*).
- Debugging the game (Deploying it and then pressing F5). Instead of asking the users to use the browser tools (which mix engine code with their own) or providing subpar tooling, it provides a full debugger that is designed from the ground up to develop Clockwork games.

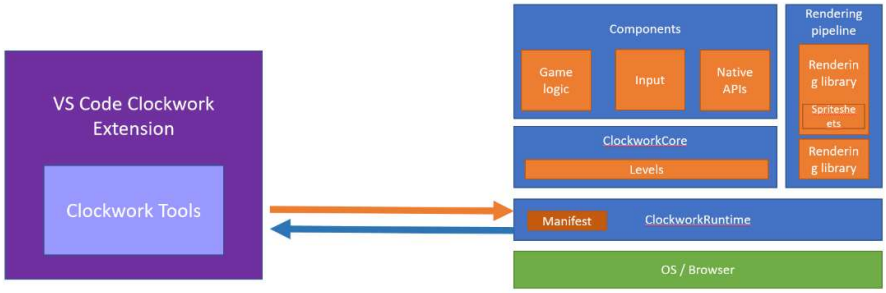


Figure 4: Diagram showing how the VS Code extension interacts with the Runtime

The debugger doesn't work as a regular JavaScript debugger, following its conventions about variable scoping, call stacks… but instead is designed specifically for Clockwork games, and has the following features:

- Instead of showing local variables, global variables, scopes and closures, as regular JavaScript debuggers do, it will just show global engine variables and the variables of the current game object, thus filtering out data that belongs to the engine or other libraries to streamline the debugging experience.
- Instead of setting breakpoints in individual lines and stepping in/out/over function calls, breakpoints will be set in event handlers, and the debugger will be able to move out/in/over engine events to provide a better understanding of what the game is doing.
- Instead of showing the function call stack, which would also display all the functions of the engine, it will show the event stack, that clearly shows which events are calling which events showing clearly the current state of the game.
- Finally, it also allows to evaluate JavaScript expressions in the scope of the current game object.

Bridges

Bridges are tools that allow developers to publish their games in different platforms, by exporting Clockwork games (.cw) to native apps:

- **Web bridge**: This bridge generates websites that host the games, so they can be played from any browser.
- **UWP bridge**: This bridge generates Windows 10 games, so they can be played on PCs, tablets, phones, Xbox…

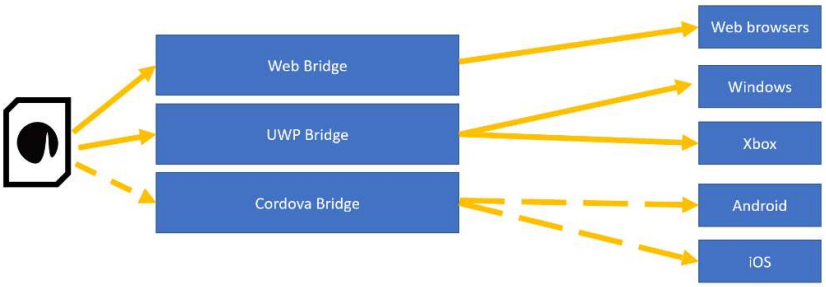


Figure 5: Diagram showing how the bridges allow to generate native apps from .cw files

DEVELOPMENT

Here is a detailed explanation of the development and inner workings of each of the parts of the Clockwork Platform described earlier:

ClockworkCore

ClockworkCore is implemented as a -relatively- lightweight (1.2k lines of code) JavaScript library that implements the following functionality:

- It manages the event loop, that runs at the frames per second specified by the user, and it is attached to a DOM element (for input and rendering purposes).
- It registers all the components that will be used in the game.
- It provides the underlying code for event handling and inheritance in components.
- It registers a rendering library used to render the game.
- It sends the state of the game to the rendering library so it can be rendered.
- It registers a list of levels that specify when and where will the objects be spawned.
- It manages the current level and allows to load new levels.
- It detects collisions between objects each frame and provides a way to perform collision queries.
- It allows to set custom loading screens that will be shown between levels until all the elements have loaded.
- It provides an API for attaching debuggers.

Components

Components are modular units that will act as blueprints for the objects that will be spawned, and they have the following properties:

- **Name**: Acts as a unique identifier.
- **Sprite**: The name of the spritesheet (that must be registered in the rendering library) that will be used to render the object.

- **Collision**: A set of objects that will specify the shapes that will act as hitboxes.
- **Vars**: A set of variables that belong to that component.
- **Events**: They are event handlers that contain all the logic of the game, implemented as functions that will be executed when some event happens. They may modify the object itself and create more events, thus altering the state of the game.
- **Inherits**: This is a single or a list of components from which the component will borrow some functionality. If a single component is specified, classical inheritance will be implemented, but if a list is used, composition will be used to create a 'virtual' component from which the current component will inherit.

Due to the characteristics of ClockworkCore, Components are expected to follow the event-driven programming paradigm, in a fashion that has been heavily inspired by SmallTalk, ActionScript and JavaScript itself. Components can be both composed and inherited, via the inheritance feature, therefore allowing to mix and match these two powerful paradigms to make sharing code as easy as possible.

Clockwork Package Manager

The Clockwork Package Manager is composed of two parts, the CLI tools (which is detailed in its own section) and the repository, a backend that hosts the packages. The backend is developed as a Node.js app hosted on Azure as a Web App, that uses the popular Express.js framework for routing, and the Tedious library for connecting to an Azure SQL database, where the packages are hosted. It also uses a small library named GearDoc, developed for this purpose, that automatically generates documentation for any Clockwork package. The web app exposes a standard REST API that returns JSON data to which any app can connect, that includes these endpoints:

- **[GET]** http://cwpm.azurewebsites.net/api/packages : This list all the available packages.
- **[GET]** http://cwpm.azurewebsites.net/api/packages/[id] : This returns all the available versions of the specified package.
- **[GET]** http://cwpm.azurewebsites.net/api/packages/[id]/[version] : This returns the content of the specified version of the specified package (as plaintext).
- **[POST]** http://cwpm.azurewebsites.net/api/packages/[id]/[version] : This stores the payload specified in the source field of the POST body as the specified package, and it is also

necessary to specify the username and password fields in the body that must correspond to the owner of the package.

- **[GET]** http://cwpm.azurewebsites.net/api/doc/[id]/[version] : This generates and returns the documentation of the specified package (in HTML format).
- **[GET]** http://cwpm.azurewebsites.net/api/developers : This lists all the registered developers.
- **[POST]** http://cwpm.azurewebsites.net/api/developers : This registers a new developer, with the username, password and email specified in the POST body.

ClockworkRuntime

ClockworkRuntime is a native app (necessarily, because of the required permissions) that allows developers to easily load, run and debug their games without needing to export them. It is currently implemented as a UWP app for Windows 10, but it could easily be exported to other desktop platforms since it is written completely in JavaScript.

It registers the cwrt:// protocol, making it possible for other tools (such as the Visual Studio Code extension) to communicate with it, and it associates with .cw files, therefore turning the task of running Clockwork games into something as easy as double clicking them.

When it receives a .cw file (which are zip files with an specific inner structure), it will unpack it and copy it to the app's local storage, plus it will download all the required dependencies of the game from the package manager if necessary (dependencies are shared between all the installed files).

When launched normally, the app will show a menu with the installed games, allowing to launch any of them as if they were a regular app. In order to debug those games, the debugger must be attached as described in the Visual Studio Code plugin.

It also implements a level editor that can be used on top of running games, but that feature is still a work in progress.

Rendering libraries

Spritesheet.js is a library designed for rendering 2D spritesheet based games. It is implemented as a standalone library that, while it implements the standard Clockwork rendering library API, can be used independently in any webpage. It can render objects using the 2D HTML5 canvas API

In order to be rendered, objects have to be associated to a spritesheet, that will specify how to draw them.

Aside from rendering the objects at the specified frames per second, it performs spatial partitions optimization in order to draw the game more efficiently.

Spritesheets in Spriteshets.js have a list of layers associated to each object state, each one made up of a list of frames, so there are four levels to consider when designing a spritesheet:

- **Frames**
    - Frames are just squares cropped from the image, and therefore are defined by
        - X position
        - Y position
        - Width
        - Height
        - A frame duration (t) must also be specified, it indicates how long will that frame be shown in the screen. A duration of 0 means that that frame will be shown as long as the animation continues.
- **Layers**
    - Layers are ordered sets of frames that are displayed one after the other. Therefore they are defined by a list of frames, and they can also be moved specifying two functions: x(t) and y(t).
- **States**
    - States are a set of layers that are displayed at the same time on the screen, overlaying if necessary. Therefore, they are just a list of layers.
- **Spritesheet**
    - Spritesheets are the set of states, plus all of their associated layers and frames, and the image from which the frames should be extracted.

For example, when animating a person walking, there could be two states: "Idle" and "Walking". Each state could be formed by several layers: "Body", "Arms", "Legs", and some layers could be reused in both states. Each layer could have several frames with the sucessive steps of the

animation. Finally, the spritesheet should contain all of them, plus a reference to "someone_walking.png".

The interface that Spritesheet.js and other rendering libraries implement is available in an annex to this document.

CLI tools

The tools are implemented as a NPM package that exports some functions that can be used to build more advanced tooling, but also exposes a simple command line interface.

It includes a starter project that will be copied and used as a template for new projects and it also can build projects into Clockwork files (.cw). Clockwork projects are folders that contain all the game assets and have a valid manifest.json at its root, and .cw files are zip files which share the same structure as the project. However, in the process of generating a .cw from a project folder, the tools perform some changes such as converting XML files to JSON.

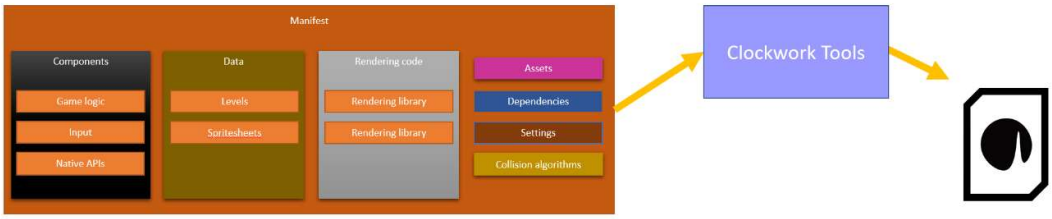


Figure 6: Diagram showing how Clockwork projects are packaged into .cw files

It uses standard HTTP requests in order to communicate with the package manager REST API and allow to list the modules, publish them, add them to the current project, update them, and register new developer accounts.

VS Code extension

This tool is implemented as an extension for Visual Studio Code, a popular open-source multiplatform code editor written in JavaScript. It is actually divided on two separate modules: the base extension, which provides most of the functionality, and the debug adapter, that allows to debug games running on the runtime.

The extension is a JavaScript module that registers several commands in the editor, createProject and buildProject, that implement their functionality by calling the functions exported

by the clockwork-tools module. It also registers two commands specifically designed for working with the clockwork runtime:

- Unlock Clockwork Runtime: Since the runtime is implemented as a UWP installed from the Windows 10 store, it is constrained by several sandboxing features. One of them is the network isolation that prevents apps from accessing the localhost (127.0.0.1), this would prevent the tools from opening a socket between VS Code and the runtime (used for deploying and debugging apps). This command runs a small PowerShell command to grant the runtime app a loopback exception.

- Deploy Clockwork project: This command packages the game, spawns a lightweight server that allows to download the .cw file and then uses the cwrt:// protocol to launch the runtime and instruct it to download and install the file from it.

Finally, it also registers an extension that allows to browse the documentation of the dependencies, by reading the manifest and loading the relevant version of the documentation in a web browser.

The debug adapter is written in TypeScript (which is transpiled to JavaScript before publishing it), and it implements both the VS Code Debug Protocol, in order to communicate with the debugger frontend, and it also opens a WebSocket (using the Socket.io library) to which Clockwork Runtime, which hosts the debugger backend, can connect. When the debugger is launched, it uses the Acorn parser to parse the code of the game and find all the event handlers, and uses them to validate and assign the breakpoints. Then it launches the runtime in debug mode via the cwrt:// protocol, which connects to the debug adapter via the WebSocket, and when the connection is established the games is launched and all the debugging commands are sent via the socket.

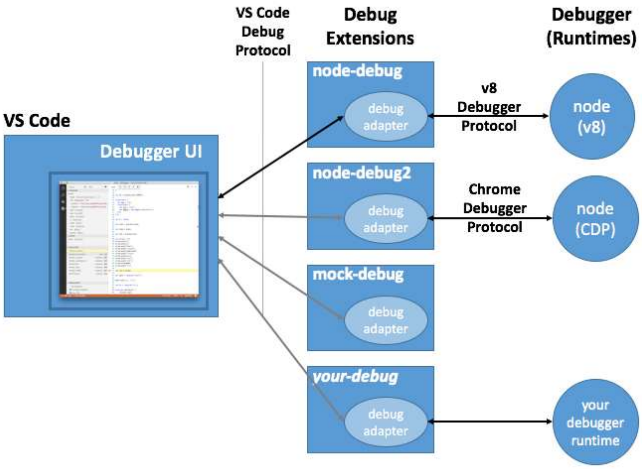


Figure 7: Diagram showing how VS Code debug extensions work [5]

The runtime will pause on the breakpoints specified in the frontend, send the global variables and the ones from the current object, update the event stack and will evaluate expressions sent by the frontend.

Bridges

WEB BRIDGE

The web bridge is a NPM package that can be used both as a command line tool and as a library since it exports the necessary function to generate static websites that implement the specified game. It extracts the content of the .cw files and combines it with a basic template that includes ClockworkCore, Spritesheet.js and a polyfill for basic runtime functionality, and then downloads all the required dependencies.

UWP BRIDGE

This bridge is a NPM package that depends on the web bridge, using it to generate a web app from the provided .cw file, and then wrapping it inside a template Visual Studio Project, replacing some web APIs with UWP ones, and finally runs MSBuild to generate a valid .appx package that can be installed in any Windows 10 device, including PC's, tablets, phones and Xbox.

INTEGRATION, TESTS AND RESULTS

In order to test the platform as a whole, a set of games and small demos have been developed as tests to prevent the appearance of bugs and regressions, as a showcase of the capabilities of the platform, and to act as code samples.

Games

THE VOID

Developed by Silvia Barbero before most of the platform was ready, this game runs on ClockworkCore and managed to win the Online Finals and the Semifinals of the videogame category of the Imagine Cup, a software development contest held by Microsoft, facing against games developed with commercial software engines.

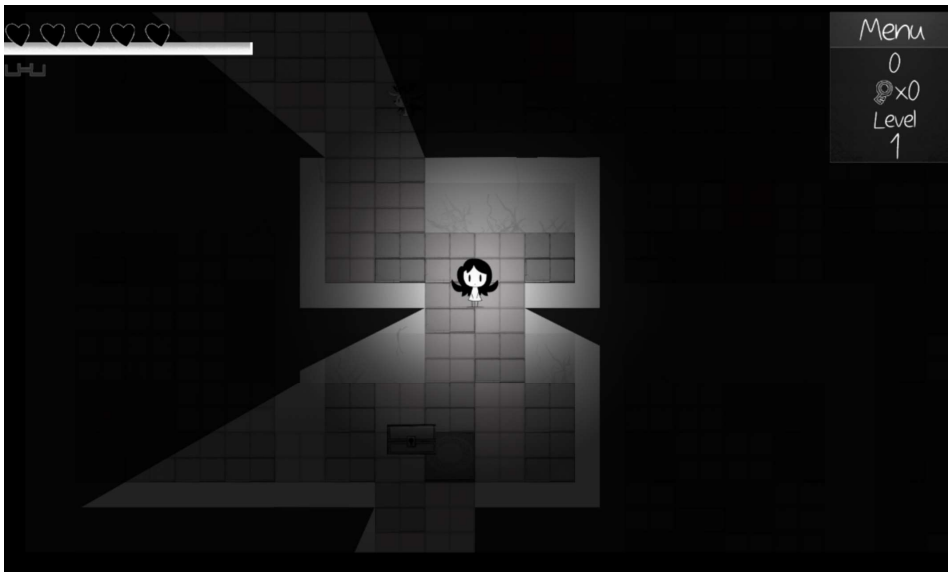


Figure 8: Screenshot of The Void

CHRONOPIRATES

This is a short game that also runs on top of Clockwork core, and it is a complete showcase of Clockwork's versatility: it includes complex procedurally generated levels, a timetravel mechanic where the user can rewind through the game creating many timelines that will replay, and even connects to a HoloLens app via a network socket to provide Mixed Reality experiences.

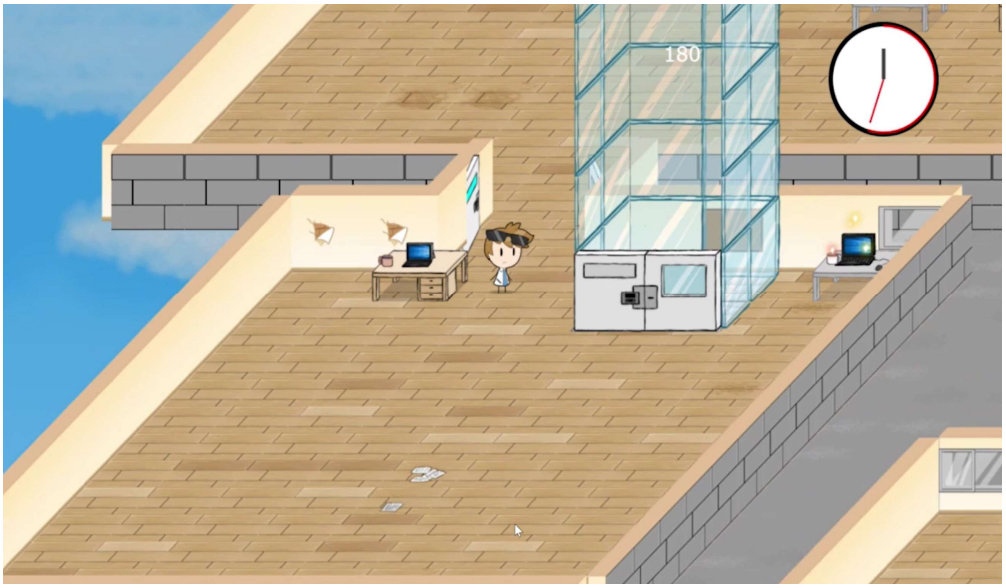


Figure 9: Screenshot of Chronopirates

TREASURE TIME

This coop platformer game is currently being developed by Silvia Barbero [4] using all the tools provided by Clockwork and is an example of the potential of the platform. With custom physics, dynamic light, beautiful maps with textures set at the start of the level depending on the terrain, many character abilities like swimming, breaking through rocks or controlling shadow, and two characters to play as to cooperate with a friend, this game runs perfectly with Clockwork.

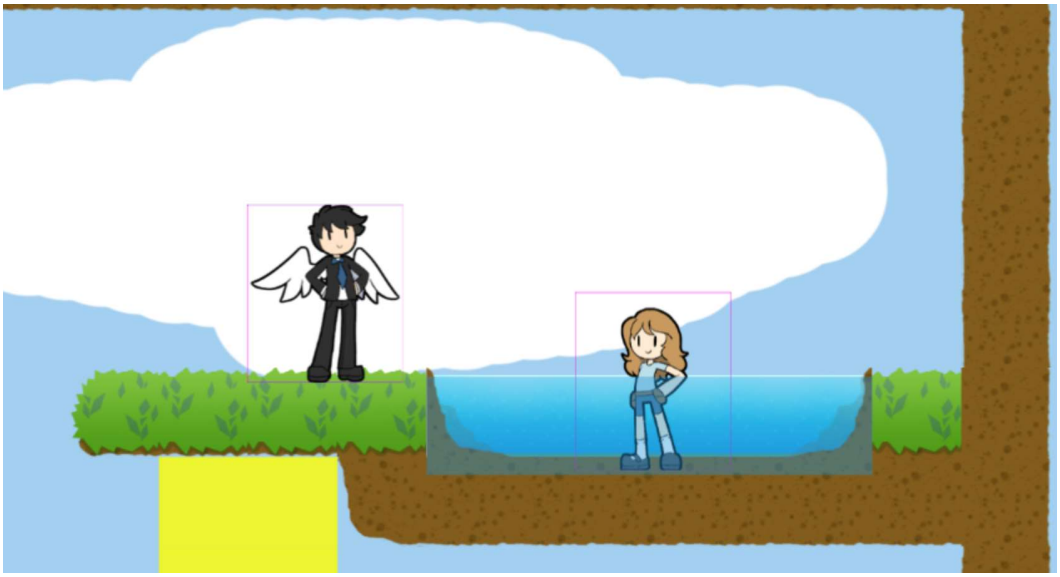


Figure 10: Screenshot of Treasure Time

ONECARD

This multiplayer card game is currently being developed to show how Clockwork can be used to easily create great multiplayer experiences easily, by sharing the same code in the server and in the clients! It uses Twitter login and uses Socket.io to connect the instances of the engine running on the clients to the one on the server, that runs another copy of the game for detecting cheats.

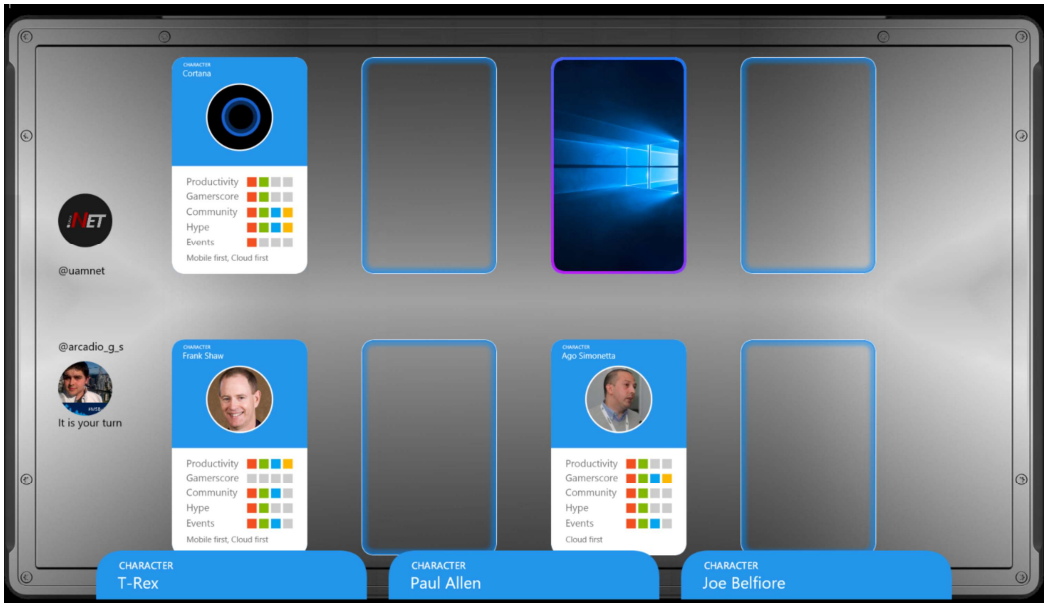


Figure 11: Screenshot of OneCard

Demos

PONG

This implementation of the classic game Pong is designed as a minimal showcase of Clockwork, to provide an easy to understand codebase developers can read to learn some of the platform features, while still implementing a whole game.

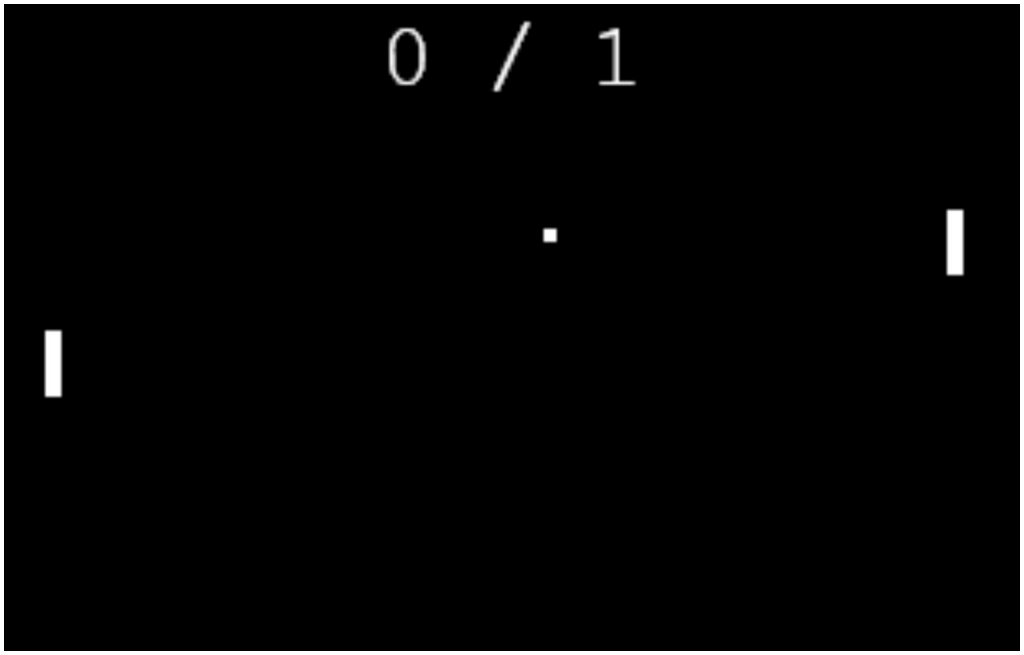


Figure 12: Screenshot of Pong

SHOOTEMUP

This is a very simple arcade game developed as a tutorial for learning how to make games with Clockwork.

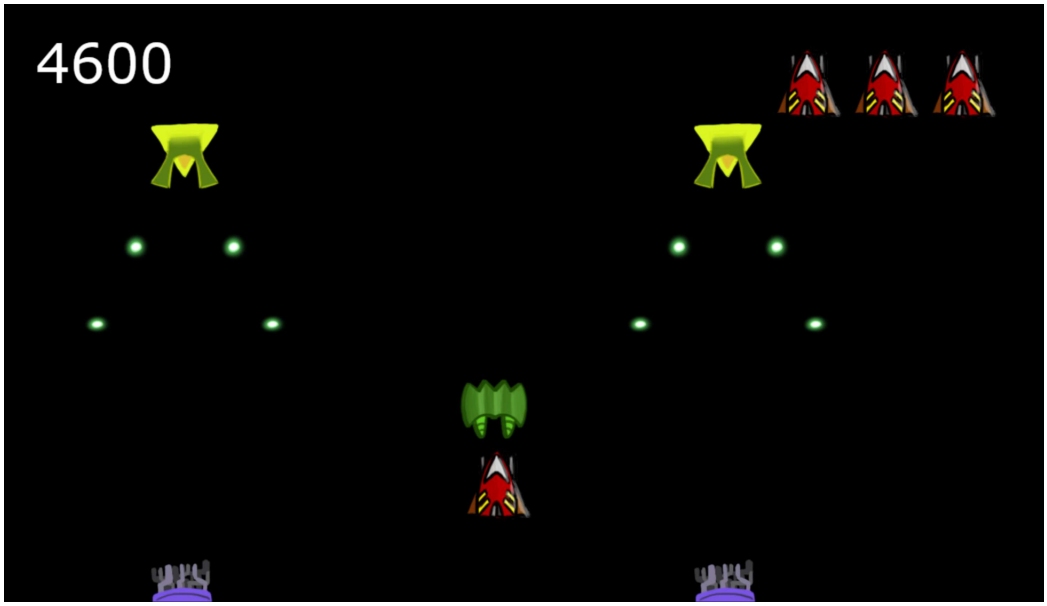


Figure 13: Screenshot of ShootEmUp

VRCOLORS

This is a very simple game that plays like 'Simon says': you must press the button with the indicated color. But there is a twist: you play in Virtual Reality, showing how 3D and even Virtual

Reality games can also be built with Clockwork! This is a showcase of how you can easily write a rendering library that connects to A-frame.

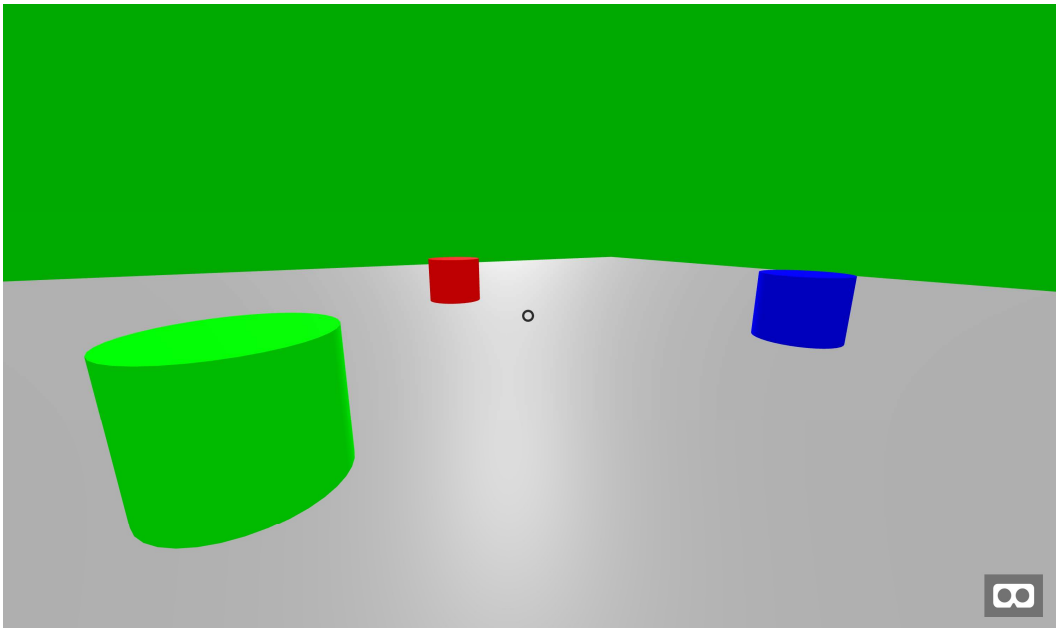


Figure 14: Screenshot of VRColors

ACCESSIBILITY

This simple combat game showcases how the ability to choose a rendering pipeline at runtime allows games to adapt to accessibility needs, providing a high contrast and a text to speech version of the game in addition to the regular one, and triggering the high contrast mode automatically depending on the PC accessibility settings.

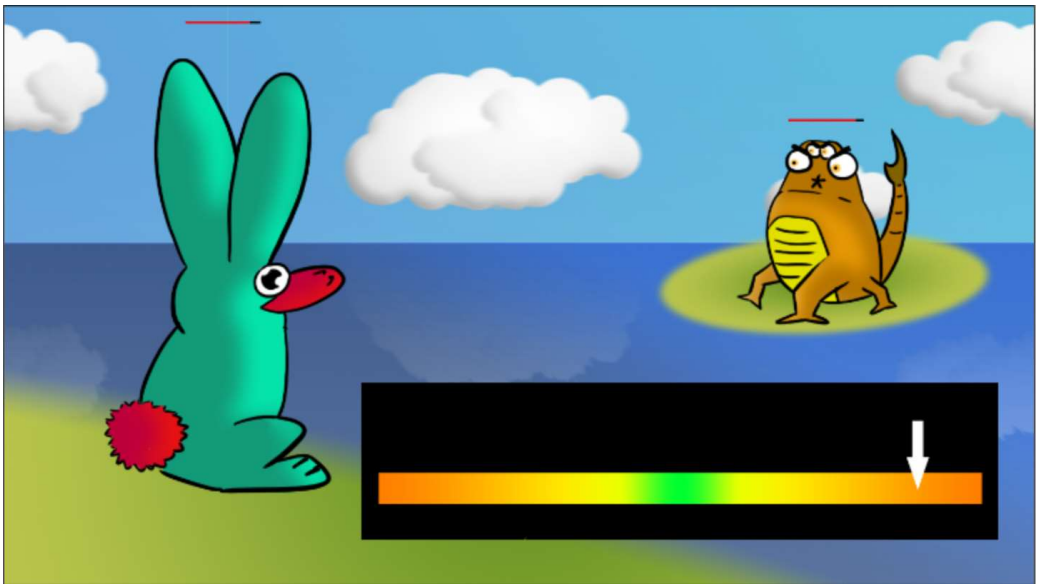


Figure 15: Screenshot of the accessibility demo in regular mode

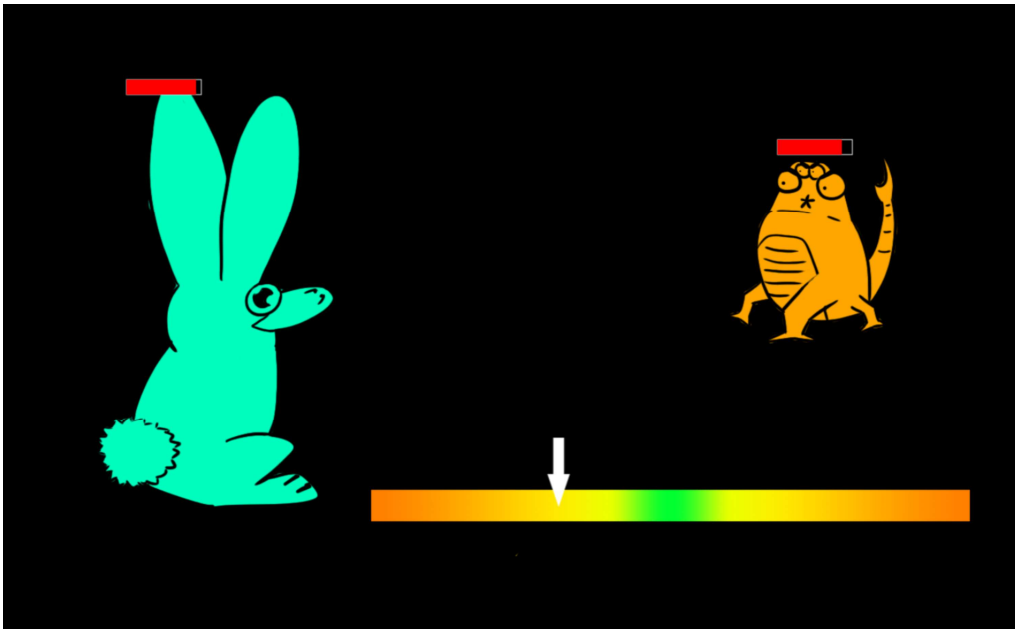


Figure 16: Screenshot of the accessibility demo in high contrast mode

DANGER ROOM

Danger Room is a game that uses the Bifrost solution (described later in this section) to deliver a multiplayer Mixed Reality experience. One player, wearing a HoloLens headset, has the ability to shoot by tapping with his finger in the air and has to destroy the bombs that will appear around him. Meanwhile, a second player will use a PC or tablet to see the position of the player in the room, thanks to the spatial mapping data sent from the device, and will place the bombs by tapping on the desired spots.

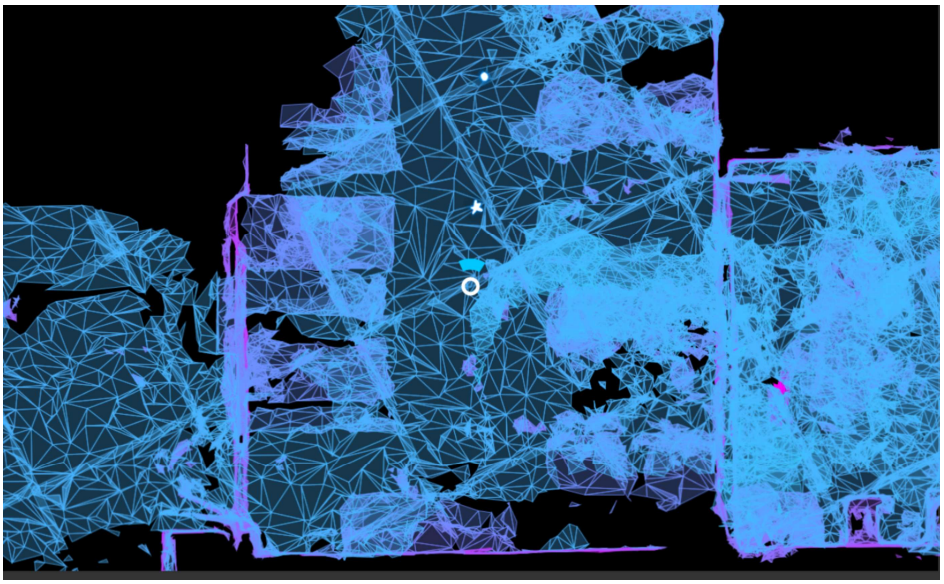


Figure 17: Screenshot of the PC app, displaying the user environment from a top down perspective

Figure 18: Screenshot of the HoloLens app, displaying a flashing bomb and recently exploded one

Packages

Most of those games are built on top of the first packages that have been published in the Clockwork Package Manager, which are:

POINTBOXCOLLISION2D

This package registers the collisions that detect when a point is inside a box.

BOXBOXCOLLISION2D

This package registers the collisions that detect when a box collides with other box.

GAMEPAD

This component allows developers to incorporate gamepad support to their game, translating the HTML5 Gamepad API to events that other components can listen to.

KEYBOARD

This component allows developers to incorporate keyboard support to their game, sending key events that other components can listen to.

MOUSE

This component allows developers to incorporate mouse support to their game, by registering a component that represents the mouse and will collide with the objects that overlap with the mouse and detect clicks.

SOCKETIO

This package registers two components, one that should be spawned in the clients and other for the server, that will translate data sent and received by the sockets to engine events.

SOCKETIOAUTH

This package registers two components, one that should be spawned in the clients and other for the server, that will provide an additional security layer on top of the socketio component by filtering the messages of the users that haven't been authenticated with Twitter before.

Rendering libraries

Finally, as a showcase of the versatility of the rendering library interface, the following rendering libraries have been developed:

A-FRAME

A-Frame is an opensource library developed by Mozilla, that builds on top of the Three.js WebGL library and allows developers to easily create Virtual Reality content for the web. An A-Frame shim has been developed, that allows Clockwork games to use A-Frame seamlessly as their rendering library (instead of Spritesheet.js) and develop VR games easily.

BIFROST

Bifrost is a solution for creating Mixed Reality games for HoloLens using the Clockwork platform. It is composed of two elements:

- Clockwork Holographic Client (Asgard): A Holographic UWP app that runs on the HoloLens and mimics the behavior of Spritesheet.js in Mixed Reality. It uses the DirectX API to render the sprites in 3D using the billboarding technique, and uses the spatial input and mapping APIs to detect natural user input and map the environment in real time.
- Heimdall: A rendering library for Clockwork that, aside from relaying the commands to Spritesheet.js to render on the current device, connects to the Clockwork Holographic Client via a TCP socket. It uses that socket to send the assets, spritesheets and commands needed to render the game, and receives the spatial mapping and user input and makes it available to the engine.

Using this solution, developers can create Mixed Reality games without any knowledge of 3D rendering, and create multi-device experiences where the same content is rendered differently on a HoloLens headset and a PC/Tablet/Smartphone.

Extension points

One of the strong points of the platform is the high number of extension points that can be used by developers to provide additional functionality and adapt the engine to their specific needs. All of these features can be completely customized:

- **Game logic**: Any game logic can by implemented by writing components that will listen to the relevant events, send their own events, and will be rendered as specified in their spritesheet.
- **Input:** Components can detect keyboard, mouse, touch, gamepad or any kind of input and translate it into events that will be consumed by other components.
- **Native APIs:** Components have full access to all the Native APIs of the platform the game is running on (HTML5 APIs on web browsers, UWP APIs on Windows 10 apps, Plugins in Apache Cordova, NPM modules in Node.js…), and via feature detection they can adapt their feature set to the current platform.
- **Rendering libraries**: Rendering libraries that follow the standard rendering library interface can be used for modifying or rendering the output of the game.
- **Rendering pipeline**: The rendering pipeline allows to chain several rendering libraries, allowing the developer to reuse and adapt existing libraries and even choose a rendering pipeline on runtime.
- **Collision algorithms**: Collider functions can be registered to allow the engine to detect any kind of collision, and even the algorithm that performs the collision detection can be tweaked for improving the performance for specific use cases.
- **Physics**: Components can be implemented that build on top of the collision detection system and implement any kind of physics, and then used in the relevant game objects via inheritance.
- **Additional tools**: Game specific tools such as level editors, animation editors, dialog editors…. Can be implemented on top of the vanilla Clockwork tooling.
- **Runtime platform**: The runtime is opensource and all the platform specific code is properly isolated, so it can be easily ported to new platforms.
- **Bridge platforms**: Using the runtime and the web bridge as reference, new bridges can be developed to export games to any new platforms.

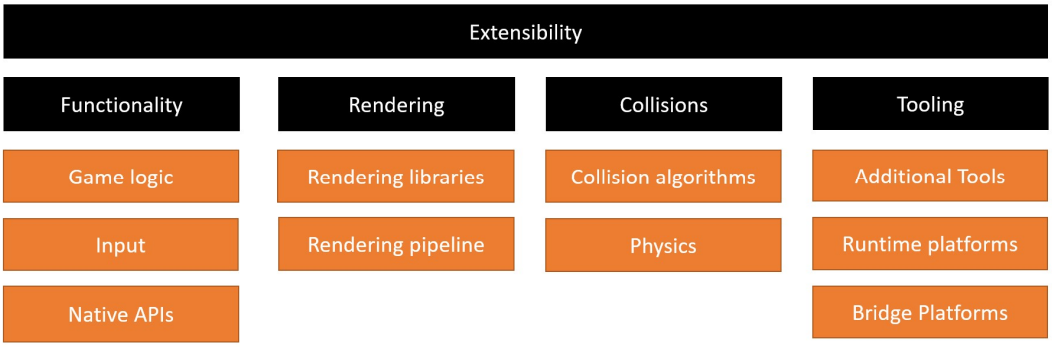


Figure 19: Classification of the extension points of the Clockwork Platform

Comparative analysis

Here is a table comparing Clockwork's key differentiators versus the existing game engines previously discussed:

Feature	**Unity**	**Phaser**	**Clockwork**
Open source		✓	✓
Built on standard web tech		✓	✓
Built-in debugger	*		✓
Built-in level editor	✓		✓
Built-in package manager	✓		✓
Support for VR,MR	✓		✓
Support for custom renderers			✓
Exports games as native apps	✓		✓
Built in mod support			✓
Runs on both client and server	**		✓

** Unity's built-in debugger runs on the Mono runtime, so it can't access some native Windows APIs*

***Unity has API's for creating and connecting to authoritative game servers, but it does not provide an easy option use a generic headless engine attached to a web server*

As the table shows, the solution developed in this project manages to provide a general-purpose, versatile and powerful engine while being 100% open and relying only in web technology. The final goal is to create a rich and open ecosystem where developers can easily share and reutilize code freely and easily without being constrained by the core platform, and are able to mix and match their own parts to easily adapt to any technology existing or to come.

CONCLUSIONS AND FUTURE WORK

Conclusions

The features of the platform make it an interesting option for the following target audiences:

- Developers that want to use web technologies but target native platforms.
- Developers that want to develop web games using an integrated set of tools and were left without many options after Flash's demise.
- Open source advocates who want to participate in an open ecosystem.
- Developers that need to implement their own rendering code (because of accessibility/artistic/technical reasons) but still want to take advantage of an existing engine and ecosystem.
- Hackers and tinkerers, developers who want to have the freedom to make headless games for a Raspberry Pi or experiment with VR games, or adapt to platform to do anything they can think of.
- Developers who want to create games that can be easily modded without having to implement their functionality themselves.
- Developers that want to develop multiplayer games and reuse code code between the server and clients easily.

Because of that, Clockwork has the potential to disrupt the status of HTML5 game development and become an attractive option for developing web games (which many forecast to have a bigger share of the whole game development industry as web apps predominate).

It is intended as the foundation for the creation of a solid platform that will empower game developers to be more productive and nourish the creation of an open ecosystem for JavaScript games. Aside from its appeal for the development of traditional 2D games, its great extensibility story make it an ideal candidate to be adopted by developers targeting emerging trends, such as virtual reality, mixed reality, accessible games, and games based on multi-device interactions or the ones that rely on modding or player-created content.

Future work

While the key parts of the platform are already developed, and published, enabling developers to start building whole games, many improvements can and will be made to boost the ecosystem and improve the platform as web technologies evolve.

These features are a work in progress:

- A revamped version of the debugger that supports stepping over/into/out in all situations no matter where the event is triggered (it currently fails for events generated outside the event loop, like input/networking).
- A live level editor that allows developers to pause running games and edit the levels on the fly.
- A rendering library that mirrors a screen to other devices automatically.
- A bridge for generating Apache Cordova apps (for iOS, Android).

Other improvements that could be done in the future would be the following:

- Turning websites generated with the web bridge into Progressive Web Apps by adding a Web App Manifest and adding a basic Service Worker, in order to make them installable and work in offline mode.
- Developing rendering shims for popular rendering libraries such as Babylon.js, Three.js and Pixi.js
- Publishing more components in the package manager that implement extra functionality that might be useful for game developers, such as menu elements, music or storage.
- Key parts of the engine, such as the collision detection logic, could be rewritten in the upcoming Web Assembly language to improve performance on those bottlenecks.

REFERENCES

[1] Gartner, "Gartner Says Worldwide Video Game Market to Total $93 Billion in 2013," October 2013. [Online]. Available: http://www.gartner.com/newsroom/id/2614915.

[2] P. a. W. Zackariasson, The Video Game Industry: Formation, Present State, and Future., New York: Routledge: Routledge, 2012.

[3] U. Technologies. [Online]. Available: https://unity3d.com/public-relations.

[4] S. B. Rodriguez, "Development of a videogame using a JavaScript engine," 2017.

[5] "Visual Studio Code documentation," [Online]. Available: https://code.visualstudio.com/docs/extensions/example-debuggers.

GLOSSARY

AAA game	A videogame with a very high development budget.
ActionScript	The programming language used for developing with Adobe Flash.
Acorn	An open-source JavaScript parser written in JavaScript.
Adobe Flash	A platform for creating rich multimedia content for the web via a browser plugin, now in disuse.
A-Frame	An open-source framework by Mozilla for building VR experiences on top of Three.js
Apache Cordova	An open-source mobile application development framework that enables developers to build applications for mobile devices with web technologies.
API	Application Programming Interface
Augmented Reality	A view of a real world environment with computer generated elements overlaid on top.
Authoritative server	In multiplayer games, a server that runs its own simulation of the game and validates the actions of the players to prevent cheating.
Azure	A cloud computing service created by Microsoft that provides varied Software as a Service, Platform as a Service and Infrastructure as a Service offerings.
Azure SQL Database	A relational database-as-a-service that uses the Microsoft SQL Server Engine.
Azure Web Apps	A fully managed (PaaS) compute platform optimized for hosting websites.
Babylon.js	A JavaScript framework for rendering 3D games using WebGL
Canvas	An HTML5 element that can be used to draw graphics via either a 2D or WebGL API.
CLI	Command Line Interface.
DirectX	A set of low level APIs for game development provided by Windows. The main one is Direct3D, which can be used to render GPU accelerated 3D graphics.

DOM	Document Object Model, the interface that treats an HTML document as a tree structure where each node is an part of the document.
Express.js	A web application framework for Node.js that takes care of routing and managing other middleware.
GitHub	A web-based Git repository, popular among open source projects.
Headless	A software running without displaying any output or accepting input.
HTML5	The latest version of the markup language used for displaying content in web browsers.
HoloLens	A self-contained, holographic computer that allows users to engage with Mixed Reality content.
IDE	Integrated Development Environment
Indie game	A videogame created without the financial support of a publisher.
JavaScript	A high-level, dynamic, multiparadigm language supported by all modern web browsers.
JSON	JavaScript Object Notation, a lightweight data-interchange format.
Mixed Reality	The merging of real and virtual worlds, where physical and virtual objects interact in real time, a hybrid between Augmented and Virtual Reality.
Modding	The act of modifying existing games by adding unofficial modules (mods) that modify its behavior or add additional content.
Mono	An open source implementation of the .NET Framework, used by Unity.
MSBuild	The build tool used for, among other things, packaging UWP apps.
Node.js	An open-source, cross-platform JavaScript runtime. It uses an event-driven, non-blocking I/O model targeted for server-side applications.
NPM	The default package manager for Node.js.
Package manager	A software tool that automates the installation of programs or libraries.
Pixi.js	A 2D rendering JavaScript rendering library that uses WebGL.
Phaser	A 2D framework for developing HTML5 games.
Polyfill	A piece of code that implements a feature in a platform or web browser that doesn’t support that feature out of the box.

PowerShell	A command line shell and associated scripting language built on top of the .NET Framework.
Progressive Web Apps	Web Apps that try to appear as native applications while being build using web technologies.
Raspberry Pi	A small single-board computer popular among hobbyists.
Rendering	Displaying an image by means of computer graphics.
REST	REpresentational State Transfer, a stateless architecture for web APIs.
Service Worker	Event-driven workers defined as a JavaScript file that are registered in a webpage and run as proxy servers between the browser and the network.
Shim	A small library that intercepts and modifies API calls.
SmallTalk	A language first introduced in 1972 that pioneered object oriented programming by using objects and messages as the basis for computation.
Socket.io	A JavaScript library for realtime bidirectional communications, using the WebSocket protocol.
Tedious	A JavaScript library that provides an implementation of the TDS protocol, which is used to interact with instances of Microsoft's SQL Server.
Three.js	A cross-browser JavaScript library used to display 3D graphics using WebGL.
TypeScript	A superset of JavaScript that adds optional static typing and class-based object programming, can be transpiled to JavaScript.
Unity	A popular cross-platform game engine.
UWP	Universal Windows Platform, an application architecture introduced in Windows 10 that allows apps to run on every Windows based device, including phones, tablets, PC's, Xbox, HoloLens, Surface Hub and IoT devices.
Virtual Reality	The simulation of a completely virtual environment that allows the user to interact with it.
Visual Studio Code	An open-source cross-platform code editor, featuring debugging support, embedded Git control, syntax highlighting and intelligent code completion, based on Electron.
Web Assembly	A new, portable, size- and load-time-efficient format suitable for compilation to the web.

WebGL	A JavaScript API for rendering 3D graphics in modern web browser using GPU acceleration .
WebSocket	A protocol that provides full-duplex communication over a TCP connection, implemented in web browsers.
XML	eXtensible Markup Language, a markup language for encoding documents.

ANNEXES

A Installation manual

In order to develop games or run the provided samples, you will need to install the tools. To do so, you will need a PC running Windows 10, and must follow these steps:

1. Download the Clockwork Runtime from the Windows 10 Store (https://www.microsoft.com/store/apps/9mt9ntllrqsr?cid=TFG)
2. Open Windows Settings (Win+I), go to *Update & Security* > *For developers* and enable developer mode.
3. Install Visual Studio Code (https://code.visualstudio.com/), launch it and install the Clockwork extension (https://marketplace.visualstudio.com/items?itemName=arcadio.clockwork). After installing it you will need to reload VS Code.
4. By default, UWP apps can't communicate with other software running on the same PC. Open VS Code and execute the unlock command (press F1 and start writing *Unlock Clockwork Runtime*) to let the Runtime and your dev tools talk to each other.
5. Install Node.js (https://nodejs.org/en/ , please make sure you have version 6 or later), and once it is installed (you might need to restart your PC) open a command prompt and install the command line tools by typing

   npm install clockwork-tools -g

   and pressing Enter.

Once you have followed these steps, you should be all set! You can check if everything has been installed correctly by:

1. In Visual Studio Code, open an empty folder and create a Clockwork project (press F1 > 'Create Clockwork Project').
2. Deploy your project to the Clockwork Runtime (press F1 > 'Deploy Clockwork Project') and click on its tile to run it.
3. If the demo game runs correctly, everything is installed correctly.

B Links of interest

The official webpage of the platform contains more information and detailed documentation: http://clockwork.js.org

The demo games can be found at: https://github.com/ClockworkDev/ClockworkDemos

The source code of all the modules of the platform is available at https://github.com/ClockworkDev/

An online demo of the Web and UWP bridges can be found at http://clockworkbridgesdemo.azurewebsites.net/

C Rendering Library Interface

This is a template that implements the standard interface that rendering libraries must follow in order to be used with Clockwork:

```
var RenderingLibrary = (function () {

    //Here is where you should put all your rendering code, which will be
private

    //And these are the public functions that the engine will use to talk to
your library
    //You can leave the ones that aren't relevant for your implementation
empty, and even send a warning via the debug handler
    return {
        setUp: function (canvas, nfps) {
            //This function receives a reference to a canvas element and the
number of fps requested
        },
        pauseAll: function () {
            //This function prevents the animation from updating (e.g
doesn't advance to the next frame on each animation)
        },
        restart: function () {
            //This function stars the 'animation logic' again, after
pauseAll is called
        },
        setCamera: function (x, y, z) {
            //This function sets the camera position
        },
        getCamera: function () {
            //This function gets the camera position
        },
        moveCameraX: function (x) {
            //This function moves the camera the specified distance in the x
axis
        },
        moveCameraY: function (y) {
            //This function moves the camera the specified distance in the y
axis
        },
        moveCameraZ: function (z) {
            //This function moves the camera the specified distance in the z
axis
        },
```

```
        loadSpritesheetJSONObject: function (newspritesheets) {
            //This function loads a list of spritesheets from an array of
JSON objects
        },
        addObject: function (spritesheet, state, x, y, z, isstatic) {
            //This function creates an object rendered by the given
spritesheet, at the given state, at the given positions and which might or
not have an static position relative to the camera
            //This function returns the object id
        },
        deleteObject: function (id) {
            //This function deletes the object with the given id
        },
        clear: function () {
            //This function removes all the objects
        },
        pause: function (id) {
            //This function prevents the animation of an specific object
from updating
        },
        unpause: function (id) {
            //This function restarts the animation of an specific object
        },
        setX: function (id, x) {
            //This function sets the x coordinate of an object
        },
        setY: function (id, y) {
            //This function sets the y coordinate of an object
        },
        setZ: function (id, z) {
            //This function sets the z coordinate of an object
        },
        setParameter: function (id, key, value) {
            //This function sets a parameter of an object
        },
        setState: function (id, state) {
            //This function sets the state of an object
        },
        setSpritesheet: function (id, s) {
            //This function sets the spritesheet of an object
        },
        sendCommand: function (command, commandArgs) {
            //This function sends a command to your library, you can use
this an extension point to provide additional functionality
        },
```

```
        setObjectTimer: function (id, t) {
            //Sets the internal time of an object
        },
        getObjectTimer: function (id) {
            //Gets the internal time of an object
        },
        setEndedCallback: function (id, callback) {
            //Sets a callback that will activate when the current animation
of an object stops
        },
        setRenderMode: function (mode) {
            //Sets a render mode, a function that will draw the buffer into
the actual canvas
            //It can be used for scaling and applying effects
        },
        setBufferSize: function (w, h) {
            //Sets the size of the internal buffer frame
        },
        getContext: function () {
            //Returns the drawing context of the canvas
        },
        chainWith: function (renderingLibrary) {
            //Chains to an instance of another rendering library, used in
'proxy' libraries (for recording, networking, perspective...)
        },
        getSpriteBox: function (spritesheet, state) {
            //Gets the bounding box of an spritesheet (the one that
encompasses all states, or just for one state if it is specified)
        },
        debug: function (handler) {
            //Turns the debug mode ON and sets a handler that will be used
to log all the errors that happen.
            //The handler will be called like this: 'handler("Something
happened");' to display warnings and errors
        },
        setWorkingFolder: function (folder) {
            //Sets the path from which assets should be loaded
        },
        getWorkingFolder: function () {
            //Returns the working folder
        }
    };
});
```

D Tutorial

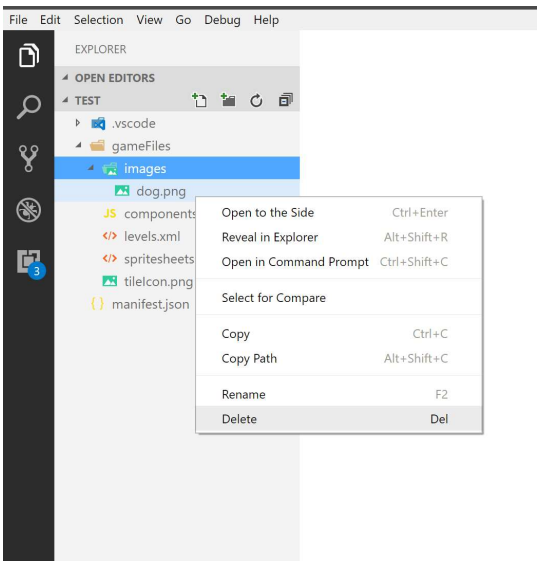
In order to teach developers how to get started developing games for the Clockwork platform, the following tutorial has been written and will be published in the official webpage:

Now, it is time to get our hands dirty and develop our first Clockwork game! Don't worry if you have no previous experience with JavaScript and/or game development, this tutorial will guide you through every step. We are going to develop the "ShootEmUp" game you can find in the provided demo games, if you get lost remember that you can find the full source code of the game at https://github.com/ClockworkDev/ClockworkDemos/tree/master/ShootEmUp and the full documentation at http://clockwork.js.org/documentation.html.

First of all, open Visual Studio Code and go to File > Open folder and select an empty folder, where you will store your project. Then, press F1 to open the VS Code command window and execute "Create Clockwork Project" (start writing the command and it will autosuggest it). It will prompt you for a project name (be creative!) and create a starter project.

You can now try to deploy your project to the ClockDwork Runtime (press F1 > 'Deploy Clockwork Project'), and if you have everything set up correctly you should see how the Clockwork Runtime launches and the game is installed (If you run this sample game, a cute dog will greet you!). If something fails, this means you might have missed some installation step, so please go back to the first section of this document and check everything.

Now, it is time to get rid of the unnecessary assets that came with the starter project. Feel free to delete dog.png and edit some files.



In levels.xml, delete this line:

```
      <object name="JakeTheDog" type="talkingDog" x="200" y="200"
></object>
```

In spritesheets.xml, delete the dog spritesheet so you are left with this:

```
<?xml version="1.0" encoding="utf-8"?>
<spritesheets>
</spritesheets>
```

In components.js, delete the dog component so you are left with this:

```
CLOCKWORKRT.components.register([
]);
```

Next, we will download the game assets. Open the folder with the file explorer and replace tileIcon.png with the image found at https://raw.githubusercontent.com/ClockworkDev/ClockworkDemos/master/ShootEmUp/gameFiles/tileIcon.png and copy the image found at https://raw.githubusercontent.com/ClockworkDev/ClockworkDemos/master/ShootEmUp/gameFiles/images/spaceships.png inside the *images* folder, with the name *spaceships.png*. Feel free to modify this pictures with your favorite image editor to give your game a unique feel!

Now, go to the manifest and make some changes:

- Change the background color to #000 (black)
- Set both the engine fps (frames per second) and the animation frames per second to 30. This will instruct the engine to execute your game logic and redraw the game 30 times each second.

We are now ready to start working on the game, so we will begin with the player's ship. We will define a spritesheet, which contains the relevant information that allows the rendering library to render it on screen. In spritesheets.xml, between the *spritesheets* tags, copy the following text:

```
  <spritesheet name="playerShip" src="images/spaceships.png">
    <states>
      <state name="Idle">
        <layer name="Idle"></layer>
      </state>
    </states>
    <layers>
```

```
    <layer name="Idle" x="0" y="0">
      <frame name="Idle"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="Idle" x="0" y="0" w="100" h="100" t="100"></frame>
  </frames>
</spritesheet>
```

This defines a spritesheet named *playerShip* that uses the images found at the specified picture, the one you downloaded earlier. It defines one possible state for that object, called Idle, which only contains a single layer, called Idle too, that itself contains a single frame called Idle (we weren't particularly inspired). This frame will draw the content of the picture that starts at 0 pixels in the x (horizontal) axis and 0 pixels in the y (vertical) axis, with a width and height of 100px. If you look at spaceships.png, you will see that this corresponds to the red vessel the player will pilot in order to save the galaxy! This frame lasts 100 ms, but this value is not really important here since we only have one frame that will be continuously looping over itself.

We have successfully defined how a spaceship looks, but that is not enough! Now we want to define how it will act inside the game, and to do so we will need to head to *components.js*. Inside the square brackets (the ones that look like this []), copy this code:

```
{
        name: "ship",
        events: [
            {
                name: "#setup", code: function (event) {
                    this.var.friction = 0.03;
                    this.var.vx = this.var.vy = 0;
                    this.var.ax = this.var.ay = 0;
                }
            },
            {
                name: "#loop", code: function (event) {
                    this.var.vx += this.var.ax;
                    this.var.vy += this.var.ay;
                    this.var.$x += this.var.vx;
```

```
                this.var.$y += this.var.vy;
                this.var.vx *= (1 - this.var.friction);
                this.var.vy *= (1 - this.var.friction);
            }
        }
    ]
}
```

Congratulations, this is your first component! A component is a piece of code that defines (totally or partially) how some objects in your game will behave. This component implements a generic spaceship, as you can deduct by its name, and has two events. Events are pieces of code that execute when something interesting happens, and these two are the most important (that is why they have a hashtag, they are keywords reserved by the engine). The #setup event will execute whenever the object is spawned, and the #loop event will execute once per frame. As you can probably deduce, we are using the setup event to initialize some values, and the loop to update them. Each object has inner variables that can be accessed and modified via the *this.var.whatever* syntax.

Here we are implementing a classic physics model: you might remember from your high school physics that speed is the derivative of position with respect of time, and acceleration is the derivative of speed with respect of time. Don't worry, we won't use any derivatives here! We just tell the object to sum the value of its acceleration in each axis (movement can be decomposed in the x and y axis through the magic of vectors :D) to the speed, and update the position according to the speed. Finally, we are slowing down the object by reducing its velocity according to its friction. You might notice the dollar signs before the x and y, you must use variables starting by $ to indicate the engine that they are necessary for drawing the object so it makes it available to the rendering library. As you can guess, knowing the position of an object is vital for drawing it on the screen!

All this is great for modelling the behavior of a generic spaceship, but we are not interested in generic spaceships! We want to implement the player ship, and to do so we are going to place a second component, right after the previous one, separated by a comma (,):

```
{
    name: "playerShip",
    sprite: "playerShip",
    inherits: "ship"
}
```

Surprise, this component is different from the previous one! We are not defining any event, but instead are using two new properties:

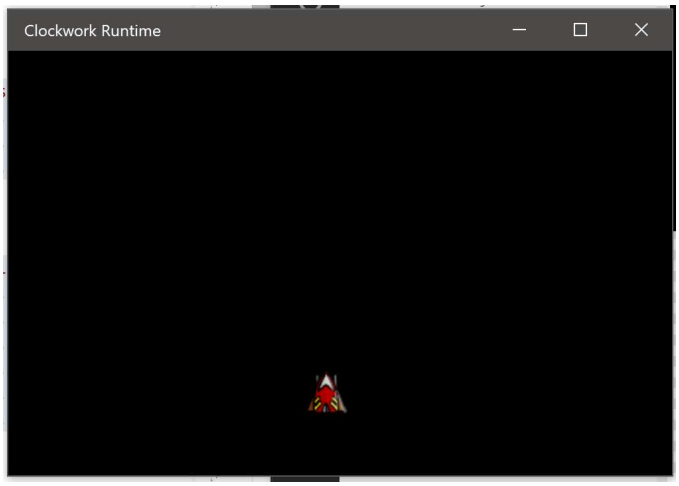
- Sprite: This sets the spritesheet used to render an object, in this case it tells it to use the spritesheet we defined earlier.
- Inherits: This sets a “parent” component from which we will inherit all its properties if we don’t say otherwise. Thanks to this, all the ship events will be added to the playerShip.

So now we have successfully defined the player ship, it is time to check our progress. Go to *levels.xml* and inside the “mainLevel” level, where you previously removed the dog, place this line:

```
<object name="playerShip" type="playerShip" x="600" y="600" ></object>
```

This will tell the engine to spawn an object called playerShip that implements the component playerShip, and place it at the coordinates (600,600).

We are now ready for takeoff! Press F1 and execute the “Deploy Clockwork Project” command. The Runtime will launch and install your game, and if everything is fine you will see your new fancy game tile. Click it to run your game, which should show the red ship still at the bottom of the empty black screen. This might not seem very impressive to others, but while implementing this you already learned most of the stuff you need to create Clockwork games.



Let’s add keyboard support to your game! And since you are now a game developer, you must follow the golden rule of development: do as little work as possible! That is why instead of dealing with keyboard input yourself, we are just going to find a package that does the work for you! (Seems like a good deal, doesn’t it?)

Press Ctrl+` inside VS Code to open a terminal inside that folder, and execute this command:

```
Clockwork add keyboard
```

It will reply

Version 1.0 of keyboard added to the dependencies

Let's find out what happened! If you go to manifest.json, you will find that someone touched your code: it is now one long messy line! First of all, you should press Shift+Alt+F to make it more readable and avoid hurting your eyes. Immediately after, you will notice something new:

```
"dependencies": {
        "keyboard": "1.0"
    },
```

A dependency has been added to the project! This will tell the Clockwork Runtime to add that piece of code to your game automatically. This seems really helpful, but we still don't know how to use it! There is an easy solution, press F1 and execute the command "Browse Clockwork package documentation", this will ask you for the package name (keyboard) and the version (1.0), and then will open the documentation for that package. By the way, that documentation is automatically generated, so it will follow the same standard style for any package. Handy, isn't it?

Anyway, if you read it you will discover that it lets you use a new component, aptly named keyboard. So, let's go to levels.xml and add it to the current level, just after the playerShip:

```
    <object name="keyboard" type="keyboard" x="0" y="0" ></object>
```

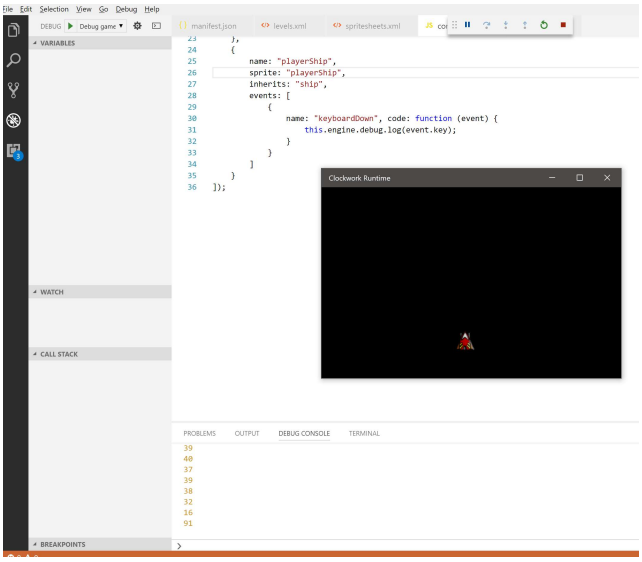
If you try deploying your game now, you will see no changes. Why is it so? Sadly, we haven't yet discovered how to read minds and predict the future (we would like it to incorporate to our roadmap though!), so we don't know *how* do you want to use that keyboard input, so you will have to do a small effort. If you read the documentation, you will see that it triggers two new events, keyboardDown and keyboardUp, which conveniently tell you which key has been pressed/lifted. The only issue is that it talks about something called the key code, and we don't know which key code correspond to each key. While a quick Bing search would reveal the answer, we are going to find out by ourselves as an excuse to try more cool Clockwork features.

Change the playerShip component so it looks like this:

```
{
        name: "playerShip",
        sprite: "playerShip",
        inherits: "ship",
        events: [
            {
                name: "keyboardDown", code: function (event) {
                    this.engine.debug.log(event.key);
```

```
                }
            }
        ]
    }
```

This will print in the debug log the code of each letter pressed. Now deploy your game again but, instead of clicking on its tile, go back to VS Code and press F5. Now go to the game and feel free to mash your keyboard (please don't hold us accountable for any physical damage you inflict to your keyboard). You will see the key codes for each letter being outputted in the debug console.



As you can imagine, the debug console is very useful for keeping track of what is happening inside your game and easily find bugs. Clockwork itself will also report common errors in your game, but remember, you must be debugging the game (F5) to see them!

Now replace that keyboardDown event by these two:

```
{
                name: "keyboardDown", code: function (event) {
                    switch (event.key) {
                        case 37:
                            this.var.ax = -1;
                            break;
                        case 38:
                            this.var.ay = -1;
                            break;
                        case 39:
```

```
                    this.var.ax = 1;
                    break;
                case 40:
                    this.var.ay = 1;
                    break;
                case 32:
                    this.do.fire();
                    break;
            }
        }
    },
    {
        name: "keyboardUp", code: function (event) {
            switch (event.key) {
                case 37:
                    this.var.ax = 0;
                    break;
                case 38:
                    this.var.ay = 0;
                    break;
                case 39:
                    this.var.ax = 0;
                    break;
                case 40:
                    this.var.ay = 0;
                    break;
            }
        }
    }
```

This is all the code you will need to process keyboard input, basically you will modify the acceleration when a key is pressed, and reset it when it is released. You will notice we are also introducing a new feature when the space bar (key code 32) is pressed. The *this.do.eventname()* instructs the current object to execute that event, but since we have not yet defined the fire event this will do nothing.

Let's see what we got! Deploy your game and you will be able to control your game using the keyboard.

It doesn't matter if your ship is able to make the Kessel Run in less than 12 parsecs if it is unable to defend against hordes of villainous aliens, so the next step will be to add some firepower.

First, add this spritesheet that defines some nice bright red projectiles:

```
<spritesheet name="playerFire" src="images/spaceships.png">
  <states>
    <state name="Idle">
      <layer name="Idle"></layer>
    </state>
  </states>
  <layers>
    <layer name="Idle" x="-50" y="-30">
      <frame name="Idle1"></frame>
      <frame name="Idle2"></frame>
      <frame name="Idle1"></frame>
      <frame name="Idle3"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="Idle1" x="400" y="0" w="100" h="100" t="50"></frame>
    <frame name="Idle2" x="400" y="100" w="100" h="100" t="50"></frame>
    <frame name="Idle3" x="400" y="200" w="100" h="100" t="50"></frame>
  </frames>
</spritesheet>
```

Now, add this component to the list:

```
{

        name: "playerFire",
        sprite: "playerFire",
        events: [
            {
```

```
                name: "#loop", code: function (event) {
                    this.var.$y -= 10;
                    if (this.var.$y < -50) {
                        this.engine.deleteObjectLive(this);
                    }
                }
            },
            {
                name: "#collide", code: function (event) {
                    this.engine.deleteObjectLive(this);
                }
            }]
    }
```

As you can see, it will use the desired spritesheet, and will move upwards 10 pixels each frame. When its position is 50 px above the upper edge of the screen, it will delete itself using the *this.engine.deleteObjectLive* function. If you have not guessed it yet, functions inside *this.* interact with the current object, while functions inside *this.engine.* interact with the whole level. It will also delete itself if it collides with something, but we will forget about that for now. Then, add the fire event to the list of events of playerShip:

```
   {
                name: "fire", code: function (event) {
                    if (Math.random() > 0.5) {
                        this.engine.addObjectLive("somePlayerFire",
"playerFire", this.var.$x + 40, this.var.$y, this.var.$z - 1);
                    } else {
                        this.engine.addObjectLive("somePlayerFire",
"playerFire", this.var.$x + 70, this.var.$y, this.var.$z - 1);
                    }
                }
            }
```

It will randomly shoot from either the left or right cannon, spawning a playerFire called somePlayerFire at the desired relative position. Deploy your game and press the space bar to see some nice fireworks!

After witnessing the firepower of your fully armed and operational battleship, it is time to give it something to shoot at! We are going to add some enemies and, as always, we are going to start adding these spritesheets to spritesheets.xml:

```
  <spritesheet name="cannonShip" src="images/spaceships.png">
  <states>
    <state name="Idle">
      <layer name="Idle"></layer>
    </state>
  </states>
  <layers>
    <layer name="Idle" x="0" y="0">
      <frame name="Idle"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="Idle" x="200" y="0" w="100" h="100" t="100"></frame>
  </frames>
</spritesheet>
<spritesheet name="triangleShip" src="images/spaceships.png">
  <states>
    <state name="Idle">
      <layer name="Idle"></layer>
    </state>
  </states>
  <layers>
    <layer name="Idle" x="0" y="0">
      <frame name="Idle"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="Idle" x="300" y="0" w="100" h="100" t="100"></frame>
  </frames>
</spritesheet>
  <spritesheet name="kamikazeShip" src="images/spaceships.png">
  <states>
    <state name="Health3">
      <layer name="Health3"></layer>
```

```
    </state>
    <state name="Health2">
      <layer name="Health2"></layer>
    </state>
    <state name="Health1">
      <layer name="Health1"></layer>
    </state>
  </states>
  <layers>
    <layer name="Health3" x="0" y="0">
      <frame name="Health3"></frame>
    </layer>
    <layer name="Health2" x="Math.random()*8-4" y="Math.random()*8-4">
      <frame name="Health2"></frame>
    </layer>
    <layer name="Health1" x="Math.random()*16-8" y="Math.random()*16-8">
      <frame name="Health1"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="Health3" x="100" y="0" w="100" h="100" t="100"></frame>
    <frame name="Health2" x="100" y="100" w="100" h="100" t="100"></frame>
    <frame name="Health1" x="100" y="200" w="100" h="100" t="100"></frame>
  </frames>
</spritesheet>
```

We have nicknamed the 3 kind of enemies ‘cannon’, ‘triangle’ and ‘kamikaze’ (Sidequest: try to guess which is which in the spritesheet and then read the coordinates of the frames to check if you were right!). The triangle and cannon spritesheets are pretty straight forward, but we are going to use a few tricks in the other: we have three different states which will be used depending on the health of the enemy to display the damage caused to the ship, and when the health is low we are adding random offsets to the position of the layer to show how its warp engine collapses into a singularity (Pro tip: this is a videogame, so just add cool stuff and then justify it with sciency words!).

Before programming the enemies, we have one problem to solve first: how can we detect when a ship collides with another? Or when a ship collides with a projectile? In order to easily let you solve that problem, Clockwork lets you define collision detectors. These will be functions that decide whether two objects with an specific shape collide, and can be implemented as you want depending on the requirements of your game. For example, you could have a game that happens in a one-dimensional universe, so to check if two objects collide you only need to know if their position in the only axis is the same, but you could also have a game that plays in 4 dimensions (if you manage to imagine that, please let us know, our brains still hurt from trying) or a game where objects collide if they occupy the same position in space and have the same color.

Luckily, we are just going to see if a point is inside a rectangle in the 2D space. Create a file called *collisions.js* inside the gameFiles folder and paste this content inside:

```
CLOCKWORKRT.collisions.register([
    {
        shape1: "player",
        shape2: "enemyFire",
        detector: function (player, enemyFire) {
            if (enemyFire.x >= player.x && enemyFire.y >= player.y &&
enemyFire.x <= player.x + player.w && enemyFire.y <= player.y + player.h) {
                return true;
            } else {
                return false;
            }
        }
    },
    {
        shape1: "enemy",
        shape2: "playerFire",
        detector: function (enemy, playerFire) {
            if (playerFire.x >= enemy.x && playerFire.y >= enemy.y &&
playerFire.x <= enemy.x + enemy.w && playerFire.y <= enemy.y + enemy.h) {
                return true;
            } else {
                return false;
            }
        }
```

```
    }
]);
```

Here we are registering two collision detectors, one for telling if an enemy fire (which will be a point) is inside the player hitbox, and another one for checking the same but for player fire and enemy hitboxes. Actually, you can see that the code for both is the same (replacing the parameter names), so why define two? That way, the engine will only check collisions of enemy fire against the player and not its fellow enemies, and vice versa, meaning it will be more efficient and you won't need to filter those false positives later. The code itself is very straightforward, it returns true if the point is at the right of the left side of the box, below the top side, at the left of the right side and above the lower side.

Now it is time to let Clockwork know you want to use this code file. Go to the manifest and update the components property:

```
  "components": [
        "collisions.js",
        "components.js"
    ],
```

You can add as many (existent) files as you want, so you could actually divide your components on several files to have your codebase more organized. For the sake of simplicity (and also because we are slightly lazy, though we won't admit that openly), we will just keep piling components in components.js.

Now is where we would tell you to create a component that inherits from ship and adds some logic for the baddies, but we have a problem here. That logic would go inside the #setup and #loop events, and if you declare them inside your enemy you will overwrite the ones from the parent (ship), but we would still like our enemy to follow the laws of physics. Don't worry, confused programmer, cause we have you covered! We are going to create a component called *kamikazeLogic*, and then create another one called *kamikazeShip* that inherits from both kamikazeLogic and ship, so it will inherit both events. This would also potentially allow us to add those "brains" to other objects aside from ships.

This are the components you need to add to components.js:

```
    {
        name: "kamikazeLogic",
        events: [
            {
```

```
            name: "#setup", code: function (event) {
                this.var.ay = 0.5;
                this.var.friction = 0.1;
                this.var.health = 3;
                this.var.$state = "Health" + this.var.health;

            }
        },
        {
            name: "#loop", code: function (event) {
                var playerShip = this.engine.find("playerShip");
                if (playerShip.var.$x < this.var.$x) {
                    this.var.ax = -1;
                } else {
                    this.var.ax = 1;
                }
                if (this.var.$y > 800) {
                    this.engine.deleteObjectLive(this);
                }
            }
        },
        {
            name: "#collide", code: function (event) {
                if (event.shape2kind == "player") {
                    this.engine.deleteObjectLive(this);
                    this.engine.addObjectLive("explosion", "explosion",
this.var.$x, this.var.$y, this.var.$z + 1);
                }
                if (event.shape2kind == "playerFire") {
                    this.engine.addObjectLive("explosion", "explosion",
this.var.$x, this.var.$y, this.var.$z + 1);
                    this.engine.do.scorePoints(100);
                    this.var.health--;
                    if (this.var.health > 0) {
```

```
                        this.var.$state = "Health" + this.var.health;
                    } else {
                        this.engine.deleteObjectLive(this);
                    }
                }
            }
        ],
        collision: {
            "enemy": [
                { "x": 0, "y": 0, "w": 100, "h": 100, "#tag":
"shipCollision" },
            ],
            "enemyFire": [
                { "x": 50, "y": 100, "#tag": "shipCollision" },
            ]
        }
    },
    {
        name: "kamikazeShip",
        sprite: "kamikazeShip",
        inherits: ["ship", "kamikazeLogic"]
    }
```

KamikazeLogic will use the setup event to set a permanent acceleration in the y axis, overwrite the friction value to limit its speed and set up some variables of its own. In order to choose which state is used to render the object (from the ones defined in the spritesheet), it sets the value of the $state variable. Each frame, it will use *this.engine.find(objectName)* to get a reference to the player and thrust left or right to get closer to it, and when it is past the lower side of the screen it will silently destroy itself.

In the #collide event, we write the code that will be executed when it crashes against something. Since this ship might crash against the player in a suicidal attack or be reached by the player fire, we have to differentiate depending on the type of colliding object, found on *shape2kind* (if you want to know more about the information received in the collision events, feel free to have a look at the documentation on our website). When it crashes with the player if will destroy itself and

display an explosion, but when is reached by the player fire it will add points to the score and decrease its health, updating its state. If its health reaches 0, it will destroy itself.

The most interesting bit is the *collision* property, which sets a collider of type enemy, that defines a bounding box that covers the ship (its coordinates are relative to the object position), and a point of type enemyfire in the front of the ship. We are defining two different colliders because we want this ship to behave both as a target for enemy fire and as a bullet itself that can damage the player.

Now we just need to add the explosion spritesheet:

```
<spritesheet name="explosion" src="images/spaceships.png">
    <states>
      <state name="explosion">
        <layer name="explosion1"></layer>
        <layer name="explosion2"></layer>
        <layer name="explosion3"></layer>
      </state>
    </states>
    <layers>
      <layer name="explosion1" x="-20" y="-5">
        <frame name="explosion"></frame>
        <frame name="empty"></frame>
        <frame name="empty"></frame>
      </layer>
      <layer name="explosion2" x="5" y="0">
        <frame name="empty"></frame>
        <frame name="explosion"></frame>
        <frame name="empty"></frame>
      </layer>
      <layer name="explosion3" x="0" y="15">
        <frame name="empty"></frame>
        <frame name="empty"></frame>
        <frame name="explosion"></frame>
      </layer>
    </layers>
    <frames>
```

```
    <frame name="empty" x="0" y="200" w="100" h="100" t="100"></frame>
    <frame name="explosion" x="0" y="300" w="100" h="100" t="100"></frame>
  </frames>
</spritesheet>
```

And the explosion component:

```
{
        name: "explosion",
        sprite: "explosion",
        events: [
            {
                name: "#setup", code: function (event) {
                    this.var.timer = 0;
                }
            },
            {
                name: "#loop", code: function (event) {
                    this.var.timer++;
                    if (this.var.timer == 10) {
                        this.engine.deleteObjectLive(this);
                    }
                }
            }
        ]
    }
```

Finally, don't forget to update the playerFire component so it has a playerFire collider:

```
    {
        name: "playerFire",
        sprite: "playerFire",
        events: [
            {
                name: "#loop", code: function (event) {
```

```
                this.var.$y -= 10;
                if (this.var.$y < -50) {
                    this.engine.deleteObjectLive(this);
                }
            }
        },
        {
            name: "#collide", code: function (event) {
                this.engine.deleteObjectLive(this);
            }
        }],
    collision: {
        "playerFire": [
            { "x": 0, "y": 0, "#tag": "shipCollision" },
        ]
    }
},
```

And add a collider to playerShip as well:

```
   collision: {
            "player": [
                { "x": 0, "y": 0, "w": 100, "h": 100, "#tag":
"shipCollision" },
            ]
        }
```

And we will now have all the necessary pieces in place, add this object to your level

```
      <object name="kamikazeShip" type="kamikazeShip" x="200"
y="0"></object>
```

Deploy your game and try to destroy the ship before it reaches you

Now we are going to add the other two kinds of enemies. They don't use any new clockwork features, they are just enemies that can be killed on one shot, don't damage the enemy on contact and shoot either one or two projectiles once in a while. To speed up things a little, we are just going to provide you the code here, so you can choose to either copy paste it blindly or read it so you can understand every detail. These are the components you need to add:

```
{
        name: "cannonLogic",
        events: [
            {
                name: "#setup", code: function (event) {
                    this.var.ay = 0.2;
                    this.var.friction = 0.1;
                    this.var.timer = 0;
                }
            },
            {
                name: "#loop", code: function (event) {
                    this.var.timer++;
                    if (this.var.timer == 100) {
                        this.var.timer = 0;
                        this.do.fire();
                    }
                }
            },
            {
                name: "fire", code: function (event) {
```

```
                    this.engine.addObjectLive("someWaveFire", "waveFire",
this.var.$x, this.var.$y + 100, this.var.$z - 1);
                }
            },
            {
                name: "#collide", code: function (event) {
                    this.engine.addObjectLive("explosion", "explosion",
this.var.$x, this.var.$y, this.var.$z + 1);
                    this.engine.do.scorePoints(200);
                    this.engine.deleteObjectLive(this);
                }
            }
        ],
        collision: {
            "enemy": [
                { "x": 0, "y": 0, "w": 100, "h": 100, "#tag":
"shipCollision" },
            ]
        }
    },
    {
        name: "triangleLogic",
        events: [
            {
                name: "#setup", code: function (event) {
                    this.var.ay = 0.2;
                    this.var.friction = 0.1;
                    this.var.timer = 0;
                }
            },
            {
                name: "#loop", code: function (event) {
                    this.var.timer++;
                    if (this.var.timer == 50) {
                        this.var.timer = 0;
```

```
                    this.do.fire();
                }
            }
        },
        {
            name: "fire", code: function (event) {
                var plasma1 = this.engine.addObjectLive("someWaveFire",
"plasmaFire", this.var.$x + 20, this.var.$y + 100, this.var.$z - 1);
                plasma1.var.vx = -1;
                var plasma2 = this.engine.addObjectLive("someWaveFire",
"plasmaFire", this.var.$x + 80, this.var.$y + 100, this.var.$z - 1);
                plasma2.var.vx = 1;
            }
        },
        {
            name: "#collide", code: function (event) {
                this.engine.addObjectLive("explosion", "explosion",
this.var.$x, this.var.$y, this.var.$z + 1);
                this.engine.do.scorePoints(300);
                this.engine.deleteObjectLive(this);
            }
        }
    ],
    collision: {
        "enemy": [
            { "x": 0, "y": 0, "w": 100, "h": 100, "#tag":
"shipCollision" },
        ]
    }
},
{
    name: "cannonShip",
    sprite: "cannonShip",
    inherits: ["ship", "cannonLogic"]
},
```

```
{
    name: "triangleShip",
    sprite: "triangleShip",
    inherits: ["ship", "triangleLogic"]
},
{
    name: "plasmaFire",
    sprite: "plasmaFire",
    events: [
        {
            name: "#loop", code: function (event) {
                this.var.$y += 4;
                this.var.$x += this.var.vx;
                if (this.var.$y > 800) {
                    this.engine.deleteObjectLive(this);
                }
            }
        },
        {
            name: "#collide", code: function (event) {
                this.engine.deleteObjectLive(this);
            }
        }],
    collision: {
        "enemyFire": [
            { "x": 0, "y": 0, "#tag": "shipCollision" },
        ]
    }
},
{
    name: "waveFire",
    sprite: "waveFire",
    events: [
        {
```

```
            name: "#loop", code: function (event) {
                this.var.$y += 4;
                if (this.var.$y > 800) {
                    this.engine.deleteObjectLive(this);
                }
            }
        }, {
            name: "#collide", code: function (event) {
                this.engine.deleteObjectLive(this);
            }
        }],
    collision: {
        "enemyFire": [
            { "x": 50, "y": 100, "#tag": "shipCollision" },
        ]
    }
}
```

And you will also need to add the spritesheets for the enemy fire:

```
<spritesheet name="waveFire" src="images/spaceships.png">
    <states>
      <state name="Idle">
        <layer name="Wave1"></layer>
        <layer name="Wave2"></layer>
        <layer name="Wave3"></layer>
      </state>
    </states>
    <layers>
      <layer name="Wave1" x="0" y="-(t%1000)/100">
        <frame name="Idle1"></frame>
        <frame name="Idle2"></frame>
      </layer>
      <layer name="Wave2" x="0" y="-(t%200)/5">
```

```
      <frame name="Idle2"></frame>
      <frame name="Idle3"></frame>
    </layer>
     <layer name="Wave3" x="0" y="-(1000-(t%1000))/90">
      <frame name="Idle3"></frame>
      <frame name="Idle1"></frame>
      <frame name="Idle2"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="Idle1" x="500" y="0" w="100" h="100" t="100"></frame>
    <frame name="Idle2" x="500" y="100" w="100" h="100" t="100"></frame>
    <frame name="Idle3" x="500" y="200" w="100" h="100" t="100"></frame>
  </frames>
</spritesheet>
   <spritesheet name="plasmaFire" src="images/spaceships.png">
  <states>
    <state name="Idle">
      <layer name="Idle"></layer>
    </state>
  </states>
  <layers>
    <layer name="Idle" x="-50" y="-50">
      <frame name="Idle1"></frame>
      <frame name="Idle2"></frame>
      <frame name="Idle1"></frame>
      <frame name="Idle3"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="Idle1" x="600" y="0" w="100" h="100" t="50"></frame>
    <frame name="Idle2" x="600" y="100" w="100" h="100" t="50"></frame>
    <frame name="Idle3" x="600" y="200" w="100" h="100" t="50"></frame>
  </frames>
```

```
</spritesheet>
```

Here you can notice a useful technique, we can use "t" (the number of milliseconds passed inside the current animation) as a variable to calculate the layer position, leading to some interesting effects.

Let's add the new ships to the level file to spawn them:

```
<object name="cannonShip" type="cannonShip" x="500" y="-100"></object>
      <object name="triangleShip" type="triangleShip" x="800" y="-
200"></object>
```

Try deploying the game and seeing how their behavior is defined by their code, each of them behaves differently and thus the player will need to follow a different strategy to defeat them.

You were probably able to defeat the enemy without taking a single hit, weren't you? Well, don't get too confident, remember that we haven't coded the lives system yet!

We are going to start adding a spritesheet that will display the number of lives you have left:

```
    <spritesheet name="livesIndicator" src="images/spaceships.png">
    <states>
      <state name="3lives">
        <layer name="live1"></layer>
        <layer name="live2"></layer>
        <layer name="live3"></layer>
      </state>
      <state name="2lives">
        <layer name="live1"></layer>
        <layer name="live2"></layer>
      </state>
      <state name="1lives">
        <layer name="live1"></layer>
      </state>
    </states>
    <layers>
      <layer name="live1" x="0" y="0">
        <frame name="ship"></frame>
```

```
    </layer>
    <layer name="live2" x="-100" y="0">
      <frame name="ship"></frame>
    </layer>
    <layer name="live3" x="-200" y="0">
      <frame name="ship"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="ship" x="0" y="0" w="100" h="100" t="100"></frame>
  </frames>
</spritesheet>
```

And writing the corresponding component:

```
{
    name: "livesIndicator",
    sprite: "livesIndicator",
    events: [
        {
            name: "#setup", code: function (event) {
                this.var.lives = 3;
                this.var.$state = this.var.lives + "lives";
            }
        },
        {
            name: "playerDamaged", code: function (event) {
                this.var.lives--;
                if (this.var.lives > 0) {
                    this.var.$state = this.var.lives + "lives";
                } else {
                    this.engine.loadLevelByID("mainLevel");
                }
            }
```

```
            }]
    }
```

When the player runs out of lives, it will call `this.engine.loadLevelByID` to reload the level. In a full game, you would load a different level containing a game over screen or the main menu.

We will also need to add a collision event to playerShip:

```
     {
                name: "#collide", code: function (event) {
                    this.engine.addObjectLive("explosion", "explosion",
this.var.$x, this.var.$y, this.var.$z + 1);
                    this.engine.do.playerDamaged();
                }
            }
```

And finally add the livesIndicator to the level:

```
      <object name="livesIndicator" type="livesIndicator" x="1200" y="50"
></object>
```

Deploy your game again and you will see that you can die under the blast of alien spaceships. Yay!

Now we only need to find a way to spawn them in waves instead of all at once. We are going to write a component that will take care of that:

```
{
        name: "enemySpawner",
        events: [
            {
                name: "#setup", code: function (event) {
                    this.var.timeline = [
                        { enemy: "kamikazeShip", x: 600, t: 70 },
                        { enemy: "kamikazeShip", x: 200, t: 200 },
                        { enemy: "kamikazeShip", x: 900, t: 200 },
                        { enemy: "kamikazeShip", x: 200, t: 500 },
                        { enemy: "kamikazeShip", x: 900, t: 600 },
                        { enemy: "kamikazeShip", x: 200, t: 700 },
```

```
                    { enemy: "cannonShip", x: 600, t: 850 },
                    { enemy: "cannonShip", x: 600, t: 1000 },
                    { enemy: "kamikazeShip", x: 200, t: 1050 },
                    { enemy: "kamikazeShip", x: 900, t: 1050 },
                    { enemy: "cannonShip", x: 200, t: 1120 },
                    { enemy: "cannonShip", x: 900, t: 1120 },
                    { enemy: "kamikazeShip", x: 100, t: 1200 },
                    { enemy: "kamikazeShip", x: 300, t: 1200 },
                    { enemy: "kamikazeShip", x: 700, t: 1200 },
                    { enemy: "kamikazeShip", x: 1100, t: 1200 },
                    { enemy: "triangleShip", x: 600, t: 1300 },
                    { enemy: "triangleShip", x: 200, t: 1400 },
                    { enemy: "triangleShip", x: 900, t: 1400 },
                    { enemy: "kamikazeShip", x: 100, t: 1410 },
                    { enemy: "kamikazeShip", x: 100, t: 1410 },
                    { enemy: "kamikazeShip", x: 1100, t: 1410 },
                    { enemy: "kamikazeShip", x: 600, t: 1410 },
                    { enemy: "cannonShip", x: 200, t: 1120 },
                    { enemy: "cannonShip", x: 900, t: 1120 },
                ];
                this.var.timer = 0;
            }
        },
        {
            name: "#loop", code: function (event) {
                var that = this;
                this.var.timeline.filter(function (event) {
                    return event.t == that.var.timer;
                }).forEach(function (event) {
                    that.engine.addObjectLive("spawnedEnemy",
event.enemy, event.x, -100, 0);
                });
                this.var.timer++;
            }
```

```
            }
        ]
    }
```

It has no spritesheet, since it will be invisible, but is has stored the list of enemies, with the position and time at which they should be spawned. Then every frame, it will update the timer, and filter that list to the ones corresponding to the current time, and for each of them (none in most of the frames), it will spawn the specified enemy at the specified x coordinate. This enemy list is an example, make your own to design unique levels!

In the levels file, delete the three enemy ships and replace them by the enemy spawner:

```
<object name="enemySpawner" type="enemySpawner" x="0" y="0" ></object>
```

Now try your game and try to make your way through your own level. We are almost done! The last thing we are going to add is a score counter so the player can brag about how many extraterrestrial enemies they have blasted. Its spritesheet will be the following:

```
<spritesheet name="score">
  <states>
    <state name="score">
      <layer name="score"></layer>
    </state>
  </states>
  <layers>
    <layer name="score" x="0" y="0">
      <frame name="score"></frame>
    </layer>
  </layers>
  <frames>
    <frame name="score" code="context.font='70px
Verdana';context.fillStyle='white';context.fillText(vars['$score'],x,y);"></
frame>
  </frames>
</spritesheet>
```

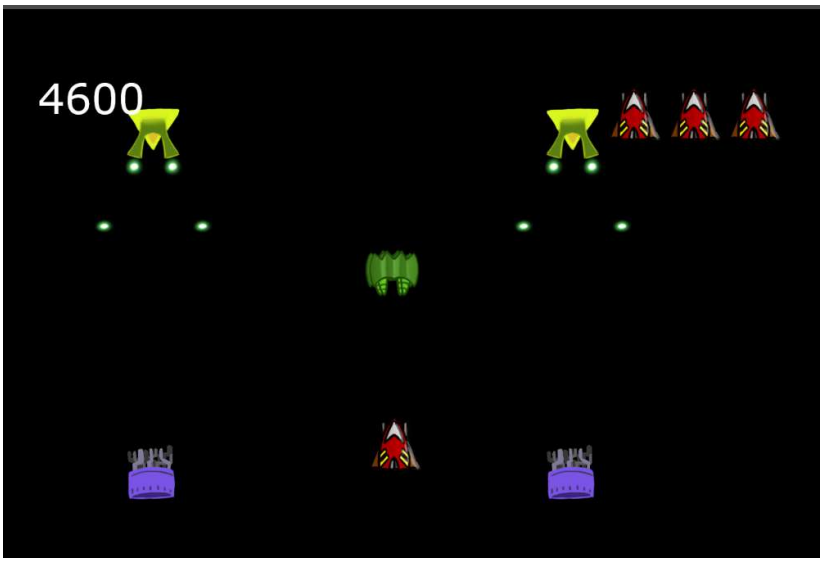
As you can see, instead of using a picture as a source, we are using a code frame, which will execute the specified instructions. In this case, it will draw the text found in the $score variable at the specified position.

Its component will be like this:

```
{
    name: "score",
    sprite: "score",
    events: [
        {
            name: "#setup", code: function (event) {
                this.var.$score = 0;
            }
        },
        {
            name: "scorePoints", code: function (event) {
                this.var.$score += event;
            }
        }
    ]
}
```

It listens to the scorePoints event produced by the enemies when they die and updates the score. Finally, let's add it to the level:

```
<object name="score" type="score" x="60" y="100" ></object>
```



Deploy your game and everything should work! Congratulations, you just made your first Clockwork game! Now you could keep adding spritesheets and components to improve the gameplay, import more dependencies such as support for gamepad or touch screens, or even change the rendering library to develop a VR game. You should definitely try uploading your .cw file to the web bridge demo (http://clockworkbridgesdemo.azurewebsites.net/web.html), so you can play in a browser and even share your game with friends!

If you are still reading, thanks for your patience! We hope this tutorial gave you a better understanding of our platform, taught you how to start developing your own games and, most importantly, you had some fun!